Efeito do Escalonamento Dinâmico no Desempenho de Processadores Super Escalares

Fernando Mauro Buleo Barbosa

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por :

Prof. Edil Severiano Tavares Fernandes, Ph.D.
(Presidente)

Prof. Cláudio Luís de Amorim, Ph.D.

Prof. Manuel Lois Anido, Ph.D.

Rio de Janeiro, RJ – Brasil

Fevereiro de 1993

BARBOSA, FERNANDO MAURO BULEO

Efeito do Escalonamento Dinâmico no Desempenho de Processadores Super Escalares [Rio de Janeiro] 1993.
XI, 76 p., 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1993)
Tese – Universidade Federal do Rio de Janeiro, COPPE.

1 - Arquiteturas Super Escalares
2 - Paralelismo de Baixo Nível
3 - Escalonamento Dinâmico
4 - Execução Especulativa
I. COPPE/UFRJ
II. Título (série).

A minha mãe

Edna

e a minha avó

Mercedes.

Agradecimentos

Ao professor Edil Severiano Tavares Fernandes, pela orientação competente e dedicada, pelo incentivo ao meu desenvolvimento acadêmico e pela sua amizade e confiança.

À professora Anna Dolejsi, pelo auxílio na aquisição de artigos e pela incansável disposição nos debates sobre algoritmos de despacho.

Ao Alberto Ferreira de Souza, pelo simulador do i860 e pelo suporte no uso do mesmo.

À Nahri, pela amizade e competência que nos tornaram companheiros inseparáveis de trabalho desde a graduação.

Ao Luís Carlos Quintela, pelo companheirismo.

A todos aqueles que de alguma forma contribuíram para a elaboração desta tese.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

Efeito do Escalonamento Dinâmico no Desempenho de Processadores Super Escalares

Fernando Mauro Buleo Barbosa

Fevereiro, 1993

Orientador : Edil Severiano Tavares Fernandes
Programa : Engenharia de Sistemas e Computação

Este trabalho descreve um modelo parametrizado de Arquitetura Super Escalar. Utilizando esse modelo básico, diversas configurações de máquina foram especificadas e simuladas. Através dos resultados produzidos por essas simulações, avaliou-se o efeito de importantes detalhes arquiteturais no volume do paralelismo de baixo nível que pode ser detectado e explorado por alguns algoritmos de escalonamento dinâmico de instruções.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

Effect of Dynamic Scheduling on the Performance of Superscalar Processors

Fernando Mauro Buleo Barbosa

February, 1993

Thesis Supervisor : Edil Severiano Tavares Fernandes
Department : Computing and Systems Engineering

This work describes a parametrized model of Superscalar Architecture. Using this basic model, several machine configurations were specified and simulated. The results obtained from these simulations were used to evaluate the effect of important architectural details on the volume of low level paralelism that can be detected and explored by some dynamic instruction scheduling algorithms.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos processadores convencionais, as instruções de máquina são executadas seqüencialmente, uma de cada vez, e a execução de uma instrução somente é iniciada quando a anterior estiver concluída. Contudo, o desempenho do processador pode ser aumentado consideravelmente se o algoritmo responsável pelo despacho de instruções for capaz de escalonar diversas instruções para serem executadas simultaneamente.

Arquiteturas Super Escalares, uma recente tendência na implementação de processadores, são exemplos típicos de máquinas que executam diversas instruções simultaneamente. A execução em paralelo de múltiplas instruções (i.e., a exploração do paralelismo de baixo nível) nessas arquiteturas é viabilizada pela presença de importantes detalhes de implementação. Unidades funcionais independentes, a presença de uma janela (no interior do processador) que armazena instruções provenientes de um mesmo programa de aplicação, e um mecanismo de previsão de desvios, são exemplos de detalhes arquiteturais usualmente incorporados pelos processadores Super Escalares.

Num processador Super Escalar, o algoritmo responsável pela escolha das instruções que serão iniciadas durante cada ciclo de máquina, pode ser implementado diretamente na sua unidade de controle. Nesse caso, o escalonamento é levado a cabo em tempo de execução, e por esse motivo ele é denominado algoritmo de *escalonamento dinâmico.*

Apesar do seu aspecto inovador, esse tipo de algoritmo de detecção

e extração do paralelismo de baixo nível, já havia sido implementado em duas máquinas comercializadas na década de 60: no modelo CDC-6600 da Control Data Corporation [THOR64] e no modelo 360/91 da IBM [TOMA67,FLYN67,DAND67, SAND67,BOLA67]. Esses dois processadores incorporam algoritmos de escalonamento que permitem a execução concorrente de várias instruções provenientes de um mesmo programa de aplicação. Adicionalmente, apesar das instruções serem executadas fora da ordem especificada originalmente pelo programador, o algoritmo de escalonamento garante a equivalência semântica do programa de aplicação.

Contudo, a tecnologia existente naquela época, tornava o custo de implementação de técnicas complexas como estas, muito elevado diante do desempenho obtido. Por esse motivo, nos anos subseqüentes, tais idéias deixaram de merecer a atenção dos pesquisadores, ficando relegadas a um segundo plano. Alternativamente, buscou-se aprimorar ou desenvolver outras técnicas que fossem economicamente mais viáveis. Idéias como *pipeline* e máquinas vetoriais foram amplamente discutidas e estudadas.

Com o decorrer do tempo, novas tecnologias foram desenvolvidas, tornando o custo do *hardware* cada vez mais baixo. Mais recentemente, com o advento da tecnologia *VLSI* e o conseqüente aumento na densidade dos circuitos, tornou-se possível encapsular em um único *chip*, recursos de *hardware* antes só encontrados em processadores de alto desempenho. Isto foi fundamental para o aparecimento de uma nova classe de máquinas que exploram os conceitos empregados nos processadores CDC-6600 e IBM-360/91 (i.e. as máquinas Super Escalares).

1.1 Máquinas Super Escalares

O termo **Super Escalar** é utilizado para descrever processadores que executam diversas instruções escalares concorrentemente. O termo também serve para diferenciar máquinas vetoriais deste tipo de arquitetura: em contraste com os processadores vetoriais, as máquinas super escalares escalonam (i.e., despacham) múltiplas operações **distintas** que serão executadas simultaneamente. Essa característica decorre da existência de diversas unidades funcionais independentes e do correspon-

dente algoritmo de despacho de instruções.

O escalonamento (durante cada ciclo de máquina) de múltiplas instruções que serão executadas em paralelo, a renomeação de registradores (*register renaming*) [KELL75] e a predição de desvios [JLEE84] são exemplos de detalhes arquiteturais que caracterizam os processadores super escalares.

Tendo em vista que a maioria das aplicações para microprocessadores são orientadas para processamento escalar, então podemos considerar as máquinas super escalares como representantes do próximo passo da evolução da arquitetura dos computadores.

O emprego do modelo de computação vetorial e a utilização de unidades funcionais implementadas segundo a técnica *pipeline*, foram importantes propostas que aumentaram substancialmente o desempenho dos processadores. A principal desvantagem dessas duas propostas consiste na limitada classe de programas de aplicação que pode ser efetivamente processada. Por exemplo, máquinas vetoriais são eficientes no tratamento daqueles problemas de computação científica em que é necessário operar com vetores constituídos por um grande número de componentes.

Por outro lado, para atingir o *throughput* máximo daqueles processadores que utilizam a técnica *pipeline* na implementação de suas unidades funcionais, torna-se necessário ativar instruções independentes continuamente. Desse modo, conforme discutido por Ramammorthy em [RAMA77], as instruções do programa de aplicação devem apresentar um elevado nível de paralelismo para que as unidades funcionais *pipelined* não sejam subutilizadas.

Nos experimentos realizados por Pleszkum (vide [PLES88]), foram verificados ganhos de 2 % a 4 % durante a execução de programas escalares (i.e., não vetoriais) em processadores com unidades funcionais *pipelined*. Diferentemente dessas propostas, arquiteturas super escalares não se destinam a um conjunto específico de problemas, mas sim para aplicações genéricas.

Embora ainda não exista uma classificação que seja amplamente usada (o que é bastante natural em áreas do conhecimento ainda incipientes), po-

demos utilizar o método de implementação do algoritmo de escalonamento de instruções, como critério para classificar as diferentes modalidades de máquinas super escalares. O algoritmo de escalonamento de um processador super escalar pode ser:

- Dinâmico
- Estático

Dizemos que o algoritmo é dinâmico quando ele é implementado diretamente pelo *hardware* subjacente. Neste caso, a detecção e exploração de paralelismo entre instruções é feito durante a execução do programa. O processador i960 da Intel [HINT89], o modelo RS-6000 da IBM [GROH90,WARR90] e o processador MC88110 da Motorola [DIEF92], são exemplos de máquinas com algoritmo dinâmico de escalonamento.

No método estático, o escalonamento das instruções que serão executadas em paralelo durante cada ciclo de máquina, é realizado antes da execução do programa de aplicação. Em outras palavras, o algoritmo de escalonamento é implementado em *software*, por exemplo, pelo módulo de geração de código do compilador. Como exemplos de processadores que utilizam algoritmos estáticos, podemos incluir as máquinas VLIW [FISH84] e o processador i860 da Intel [INTE89].

Ao compararmos a eficácia dos dois métodos de implementação do algoritmo de escalonamento, podemos verificar que para aqueles programas de aplicação em que há pouca informação (em tempo de compilação) acerca do paralelismo das instruções, as máquinas com mecanismo dinâmico são mais eficientes: levando em conta que nessas situações o compilador terá que optar por uma decisão "segura" então o código gerado nem sempre será capaz de explorar ao máximo o paralelismo existente no programa de aplicação. Uma outra vantagem apresentada pelo escalonamento dinâmico refere-se a tendência, verificada nos últimos anos, de redução no custo do *hardware* e o aumento observado no custo do desenvolvimento do *software*.

Examinando a organização dos processadores super escalares da atualidade, verificamos que eles empregam soluções utilizadas em processadores anteriores. Clássicos exemplos dessas soluções incluem o *scoreboard* e a janela de instruções do CDC-6600 [THOR64], e as *reservation stations* do IBM-360/91 [TOMA67]. A arquitetura *decoupled* [SMIT82] e a organização da máquina HPS [PATT85], são

exemplos de outras soluções que foram adotadas durante o projeto de diversos processadores super escalares comercialmente disponíveis.

Trabalhos relacionados com o despacho de múltiplas instruções incluem: propostas de algoritmos de escalonamento [TJAD70], [TJAD73], [KELL75], [AUHT85] e [ACOS86]; experimentos avaliando o desempenho de processadores super escalares [WEIS84], [PLES88], [CHAN91] e [BUTL91]; um levantamento (*survey*) abordando as técnicas de seqüenciamento de instruções é apresentada por Krick e Dollas em [KRIC91].

Apesar do grande volume de estudos realizados, só mais recentemente é que alguns processadores super escalares com algoritmo de escalonamento dinâmico, tornaram-se comercialmente disponíveis. Esse é o caso dos processadores i960 da Intel (1989), RS-6000 da IBM (1990), e MC88110 da Motorola (1992).

1.2 Motivação

O nosso interesse em estudar as principais técnicas de exploração do paralelismo de baixo nível, motivou o desenvolvimento dos experimentos aqui apresentados. A principal contribuição desta tese refere-se a avaliação do efeito de alguns algoritmos dinâmicos de escalonamento de instruções no desempenho de processadores super escalares hipotéticos. Após termos avaliado a eficácia desses algoritmos, decidimos concentrar nossos esforços no estudo do algoritmo de Tomasulo. Uma breve descrição desse algoritmo segue-se.

Quando da especificação do modelo 360/91 da IBM [SAND67], os projetistas decidiram incluir unidades funcionais separadas para o processamento de operações em ponto flutuante. Essas unidades são controladas por um algoritmo de escalonamento —diretamente implementado no *hardware*— descrito por Tomasulo em [TOMA67] e que é conhecido como "Algoritmo de Tomasulo." Através de um esquema de rotulação de registradores e de um barramento de dados comum (*Common Data Bus - CDB*) a todas as unidades funcionais, o algoritmo de Tomasulo é responsável pela ativação de operações em ponto flutuante que podem

ser executadas simultaneamente. Com o objetivo de reduzir o tempo de processamento, o algoritmo permite que operações iniciem (e terminem) fora de ordem pois ele preserva a equivalência semântica do programa de aplicação.

O conceito de dispositivos virtuais, denominados *reservation stations*, foi proposto e explorado por esse algoritmo de escalonamento: ao invés das instruções serem despachadas diretamente para os respectivos dispositivos funcionais, o algoritmo de Tomasulo transfere-as para esses dispositivos virtuais. É a partir da *reservation station* que uma operação com reais é executada.

As principais razões que motivaram o emprego do algoritmo de Tomasulo no nosso estudo seguem-se:

- sua eficácia na coordenação das atividades das máquinas super escalares;
- a simplicidade do algoritmo;
- a abrangência do algoritmo na exploração do paralelismo entre instruções.

1.3 Metodologia Adotada

No nosso trabalho, a estratégia adotada foi a de avaliar, através de simulações, a eficiência do algoritmo de Tomasulo no desempenho de arquiteturas constituídas de múltiplas unidades funcionais. Ao invés de definirmos uma nova arquitetura, decidimos utilizar uma família de máquinas derivada de um processador super escalar real, isto é, o processador i860 da Intel Corporation [INTE89]. Dessa forma, nosso modelo de máquina básica reconhece o repertório de instruções do i860. Os tipos de dados e de unidades funcionais desse processador também foram mantidos no nosso modelo.

O roteiro adotado durante nossos experimentos, pode ser grupado em três fases:

(a) Especificação e avaliação de alguns algoritmos de escalonamento dinâmico;

(b) Emprego de técnicas de balanceamento de arquiteturas super escalares;

(c) Avaliação do efeito de sofisticadas técnicas (despacho múltiplo e execução especulativa) no desempenho do nosso modelo básico de arquitetura super escalar.

Inicialmente especificamos alguns algoritmos de escalonamento dinâmico. Empregando esses algoritmos durante a interpretação de uma bateria de programas de teste (*benchmark*), observamos o efeito de diversos detalhes de implementação no desempenho do modelo básico. Como conseqüência dessas observações, decidimos adotar o algoritmo de Tomasulo, pois ele foi o que apresentou o melhor desempenho.

Com o objetivo de generalizar a atuação do algoritmo de Tomasulo, decidimos acrescentar no nosso modelo um outro tipo de unidade: o controlador de memória. Unidades desse tipo são responsáveis pela execução de instruções de transferência de dados entre os registradores e a memória principal (ou entre a memória *cache*). Empregado originalmente para controlar a unidade de ponto flutuante do IBM-360/91, o algoritmo de Tomasulo foi estendido a todos os tipos de unidade no nosso modelo básico.

Os programas de nossa bateria de testes (*benchmark*), codificados na linguagem C, foram traduzidos para o código de máquina do i860 por um compilador comercial. Em seguida, o código foi introduzido em um interpretador do processador i860, obtendo-se como resultado da interpretação os respectivos arquivos contendo *traces* da execução dos programas de teste. Finalmente, os *traces* foram processados por um simulador parametrizado do modelo de máquina.

Durante essas simulações, diversas medidas de desempenho foram realizadas. Através dessas medidas, foi possível avaliar o comportamento das diversas configurações do modelo de máquina, determinando-se dessa forma, a contribuição oferecida por cada dispositivo funcional no desempenho do processador.

Analisando os resultados obtidos especificamos uma configuração de máquina balanceada, usando como critério nesse caso, a relação entre o custo e o desempenho. Exemplos de medidas realizadas incluem: taxa de aceleração, nível de ocupação de cada recurso, porcentagem do tempo de execução em que um certo

número de recursos do mesmo tipo foi utilizado, ficou aguardando por operandos etc.

Nossos experimentos culminaram com a avaliação daquelas técnicas de exploração de paralelismo de baixo nível que foram desenvolvidas mais recentemente e que têm sido adotadas por arquiteturas super escalares comerciais, i.e., o despacho simultâneo de múltiplas instruções e a execução especulativa de instruções.

1.4 Organização do Texto

Essa tese está organizada em seis capítulos. O Capítulo 2 apresenta uma descrição do algoritmo de Tomasulo. O Capítulo 3 descreve as diversas alternativas de máquinas utilizadas pelos nossos experimentos. Os programas da bateria de testes (*benchmark*) e o método de simulação que utilizamos são tópicos abordados no Capítulo 4. No Capítulo 5 apresentamos os resultados desses experimentos e sua análise. As principais conclusões do trabalho estão relacionadas no Capítulo 6.

Capítulo 2

O Algoritmo de Tomasulo

As unidades de ponto flutuante do processador IBM-360/91 são controladas por um algoritmo de escalonamento dinâmico de instruções. Em homenagem ao seu criador, esse algoritmo é denominado "Algoritmo de Tomasulo" (vide [TOMA67]). Capaz de escalonar instruções que podem ser executadas em paralelo e fora da ordem original, o algoritmo de Tomasulo trata os conflitos no uso de recursos, de uma maneira muito eficiente. Este tipo de tratamento desempenha um papel importante na coordenação das atividades que ocorrem num ambiente com elevado nível de paralelismo, como é o caso das arquiteturas Super Escalares da atualidade.

O algoritmo define o conceito de **escalonamento associativo** de instruções. Graças à existência de um *Common Data Bus (CDB)*, de *reservation stations (RS)* e de um esquema de rotulação, a atividade de escalonamento de instruções permanece em operação mesmo na presença de dependência de dados entre instruções. O escalonamento só é interrompido quando não há mais *reservation stations* livres. Neste caso, o escalonamento será reativado tão logo seja liberada uma *reservation station.* A seguir, apresentamos uma breve descrição deste algoritmo.

2.1 O Escalonamento Associativo

2.1.1 As Reservation Stations

Em vez de despachar as instruções diretamente para as unidades funcionais, o algoritmo de Tomasulo transfere as instruções para unidades virtuais, denominadas *reservation stations*. As *reservation stations* atuam como *buffers*, armazenando os parâmetros das instruções (código de operação e operandos) além de algumas informações de controle. Cada *reservation station* é capaz de armazenar dados referentes a uma única instrução. Uma vez despachada (i.e., transferida) para uma *reservation station*, a instrução fica armazenada nesta unidade virtual, sendo descartada somente após o término da operação correspondente. Associada a cada unidade funcional existe uma ou mais *reservation stations*. A utilização de *reservation stations* permite reduzir o número de réplicas de unidades funcionais, tornando mais baixo o custo de implementação do projeto. Para ilustrar a utilização das *reservation stations* vamos considerar o seguinte fragmento de programa:

$$R_0 \leftarrow R_0 \times R_1$$

$$R_2 \leftarrow R_2 + R_0$$

$$R_3 \leftarrow R_3 + R_4$$

No trecho de programa, cada R_i é um dos registradores de um processador que possui unidades funcionais separadas para adição e multiplicação, uma unidade de cada tipo. Se a máquina não possuísse *reservation stations*, então as instruções seriam transferidas diretamente para as unidades funcionais.

As instruções do fragmento de programa seriam escalonadas da seguinte forma: a instrução de multiplicação iria para a unidade funcional de multiplicação e a primeira instrução de soma para a unidade de adição. Uma vez que só há uma unidade de adição, a segunda instrução de soma não poderia ser despachada pois não há unidade de adição disponível. Por essa razão, o processo de escalonamento seria interrompido. Se fosse possível despachar a terceira instrução, sua execução poderia ser iniciada imediatamente pois ela não apresenta dependências de

dados com nenhuma das instruções anteriores. Quando são freqüentes as situações em que a execução de instruções é atrasada pela ausência de recursos disponíveis, o desempenho pode ser comprometido gravemente.

Uma alternativa para resolver este problema seria aumentar o número de unidades de adição, garantindo assim, que na maioria dos casos existirá pelo menos uma unidade livre para receber a instrução. Contudo, a replicação de recursos sofisticados, como é o caso das unidades funcionais, aumenta o custo do projeto. A situação torna-se mais grave quando os recursos replicados são subutilizados. Isto pode ser observado no trecho de programa apresentado anteriormente. Uma vez incrementado o número de unidades de adição, o processo de escalonamento não seria interrompido por falta de unidades funcionais livres após termos despachado a primeira instrução de soma. Porém, por apresentar uma dependência de dados verdadeira (o registrador R_0 é usado como destino pela primeira instrução e como fonte pela segunda), a primeira soma só poderá ser executada após o término da multiplicação. Assim, teríamos uma unidade de adição ociosa, desempenhando durante algum tempo, a função de um *buffer*.

O emprego de *reservation stations* evita este desperdício, pois elas resultam da separação das funções de *buffer*, das funções de execução. Por essa razão, é importante ressaltar que para aumentar a taxa de instruções despachadas por ciclo de máquina (e conseqüentemente a taxa de aceleração dos programas de aplicação), não precisamos necessariamente aumentar o número de pares (*reservation stations, unidades funcionais*). A escolha (criteriosa) do número de *reservation stations* que ficarão associadas a cada tipo de unidade funcional é um importante parâmetro arquitetural: através dessa escolha é possível aumentar a taxa de instruções despachadas sem que tenhamos que prover um igual número de unidades funcionais.

2.1.2 O Esquema de Rotulação

Associado a cada registrador de ponto flutuante do IBM-360/91 existe um bit de ocupação e um campo de rótulo. O bit de ocupação indica se o valor armazenado no

registrador é válido ou não. O conteúdo do registrador torna-se inválido quando uma instrução escalonada anteriormente irá atualizá-lo. Neste caso, o bit de ocupação conterá o valor '1' e o campo de rótulo a identificação da *reservation station* armazenando tal instrução. Quando o conteúdo de um registrador for válido, o bit de ocupação é igual a '0'.

Toda vez que uma instrução for transferida para uma *reservation station*, os bits de ocupação dos registradores que atuarão como fonte da instrução são examinados. Se eles forem iguais a '0', o conteúdo do registrador é copiado na *reservation station*. Caso contrário, o algoritmo de Tomasulo copia o campo de rótulo na *reservation station*. Já o bit de ocupação do registrador que será utilizado para armazenar o resultado da operação (registrador destino) receberá o valor '1'. O rótulo (i.e., a identificação) da *reservation station* que armazena tal instrução é copiado para o campo de rótulo do registrador destino. A execução de uma instrução (armazenada na *reservation station*) é iniciada quando as seguintes condições forem satisfeitas:

- Existe uma unidade funcional livre;
- Os operandos fonte necessários já se encontram na *reservation station*

2.1.3 O Common Data Bus (CDB)

O *CDB* é um barramento que interconecta os componentes responsáveis pelas operações com reais do modelo 360/91: unidades funcionais, banco de registradores e *reservation stations*. Através desse barramento os resultados das operações são propagados para os outros componentes controlados pelo algoritmo.

Quando da conclusão de uma operação, a unidade funcional requisita o uso do *CDB*. Uma vez atendida, ela transmite (através do *CDB*) o resultado gerado em conjunto com o rótulo da *reservation station* que armazenava a instrução recém concluída. Em seguida, o banco de registradores e as *reservation stations* comparam este rótulo com o conteúdo de seus campos de rótulos. Se os rótulos forem iguais, o resultado transmitido através do *CDB* é transferido para o respectivo registrador e para os campos de operando das *reservation stations* que estavam aguardando por

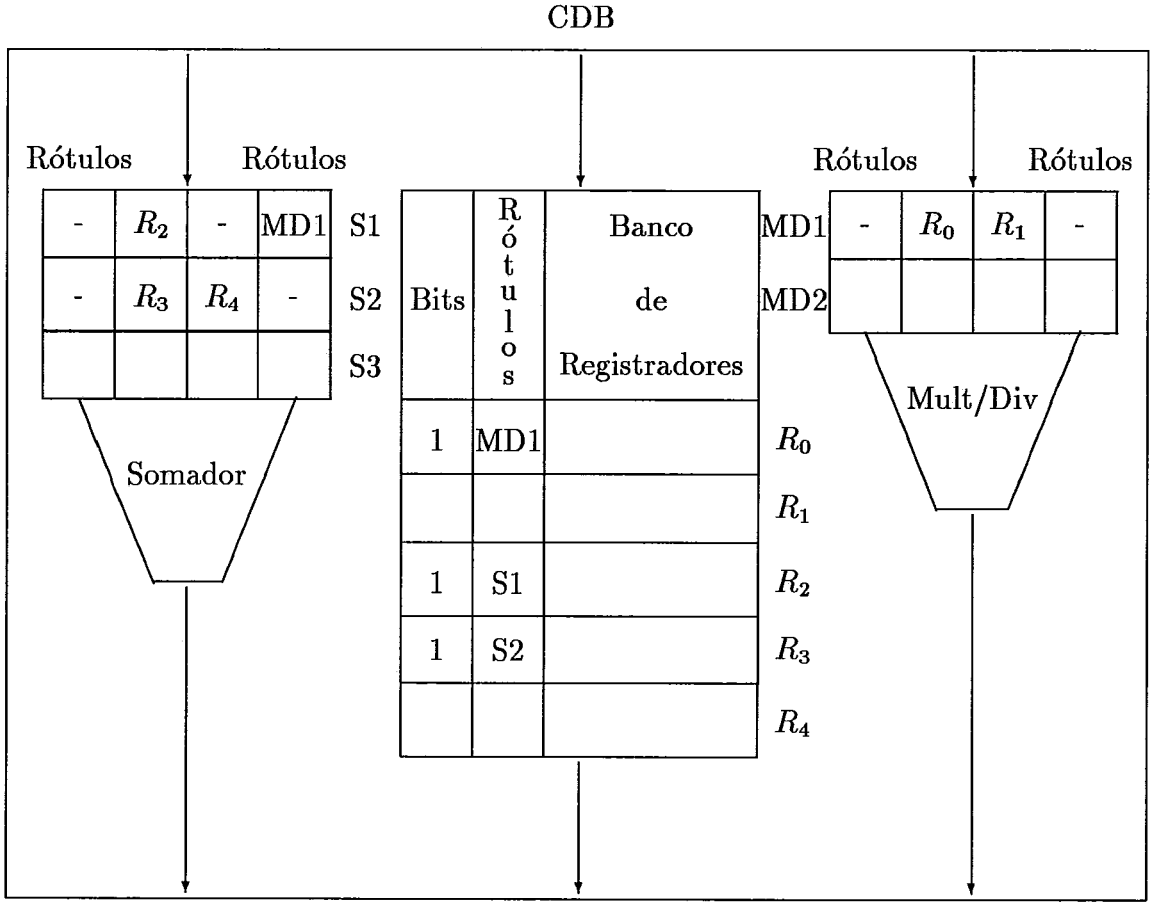

Figura 2.1: Esquema Simplificado da Unidade de Ponto Flutuante do IBM-360/91

esse resultado. O bit de ocupação do registrador cujo conteúdo foi alterado passa para '0', indicando que o valor armazenado nele é válido.

A Figura 2.1 apresenta um esquema simplificado da unidade de ponto flutuante do IBM-360/91. O conteúdo dos campos de registradores e das *reservation stations* representam o estado da máquina após o escalonamento do fragmento de programa apresentado na Seção 2.1.1 assumindo que, inicialmente, o conteúdo de todos os registradores eram válidos e todas as *reservation stations* estavam disponíveis.

2.2 Tratamento de Dependências Verdadeiras

Com o objetivo de ilustrar a atuação do algoritmo quando em presença de instruções apresentando dependência de dados verdadeira, usaremos novamente o trecho de programa apresentado na Seção 2.1.1 e da Figura 2.1.

A primeira *reservation station* associada à unidade de multiplicação/-divisão (*reservation station MD1*) foi escalonada para receber a instrução de multiplicação. Conforme ilustrado na figura, a *reservation station MD1* armazena o conteúdo dos registradores R_0 e R_1 pois eles atuam como operando fonte dessa instrução e seus bits de ocupação indicavam que seus conteúdos eram válidos. Por isso, os correspondentes campos de rótulo de *MD1* estão vazios. R_0 foi especificado como registrador destino da primeira instrução. Por esse motivo, o correspondente campo de rótulo no banco de registradores recebeu a identificação da *reservation station MD1* enquanto que o correspondente bit de ocupação recebeu o valor '1'. Desta forma, ficou indicado que a instrução armazenada na *reservation station MD1* irá atualizar o conteúdo do registrador R_0. Como todos os operandos fonte necessários já se encontram na *reservation station* e a unidade funcional de multiplicação/divisão está livre, a execução desta instrução pode ser iniciada logo após o despacho.

A segunda instrução foi transferida para a *reservation station S1*. Tendo em vista que R_2 atua como operando fonte e destino desta instrução, então o conteúdo de R_2, por ser válido, foi transferido para um dos campos de operando de *S1*, enquanto que a identificação desta *reservation station* foi armazenada no campo de rótulo de R_2, e o bit de ocupação de R_2 recebeu o valor '1'. Assim, ficou indicado que o conteúdo do registrador R_2 será atualizado pela instrução armazenada na *reservation station S1*. Com relação ao segundo operando da instrução (o registrador R_0), ao verificar que o bit de ocupação correspondente é igual a '1', o algoritmo transferiu para o segundo campo de rótulos de *S1* a identificação da *reservation station* responsável pelo cálculo do novo valor de R_0 (*reservation station MD1*). Esta identificação estava armazenada no campo de rótulo de R_0. Desta forma, ficou indicado que *S1* obterá tal operando do *CDB*, quando ele for propagado pela instrução armazenada em *MD1*. A instrução armazenada em *S1* não poderá ser executada imediatamente após seu despacho, pois terá que aguardar na *reservation station* até que todos os operandos fonte estejam disponíveis.

A terceira instrução foi transferida para a *reservation station S2*. O conteúdo dos registradores que atuam como operando fonte (R_3 e R_4) foi transferido para *S2* pois ambos são válidos. Para indicar que a instrução armazenada em *S2* irá

Eventos	*Reservation Stations*				*Registradores*					
	MD1		*S1*		R_0		R_1		R_2	
	Rot	*Rot*	*Rot*	*Rot*	*Bit*	*Rot*	*Bit*	*Rot*	*Bit*	*Rot*
Despacho Multiplicação	-	-	-	-	1	*MD*1	0	-	0	-
Despacho Adição	-	-	-	*MD*1	1	*MD*1	0	-	1	*S*1
Término Multiplicação	-	-	-	-	0	-	0	-	1	*S*1
Término Adição	-	-	-	-	0	-	0	-	0	-

Figura 2.2: Efeito do Tratamento de Dependências Verdadeiras de Dados

atualizar o conteúdo do registrador R_3, o bit de ocupação correspondente recebeu o valor '1' e o campo de rótulo recebeu a identificação de *S2*. Como todos os operandos fonte estão disponíveis em *S2* e a unidade funcional de adição está livre, esta instrução pode ser executada logo após ser despachada.

As instruções do exemplo ilustram o comportamento do escalonamento associativo na presença de uma dependência verdadeira: o registrador destino da primeira instrução é usado como operando pela segunda. A despeito desta situação adversa, o algoritmo continua escalonando as instruções subseqüentes.

Ao fim da execução da multiplicação, o novo valor de R_0 será propagado, junto com a identificação de *MD1*, para as *reservation stations* e para o registrador destino (no nosso exemplo, para *S1* e R_0 respectivamente). Após comparar o rótulo propagado pelo *CDB* com o conteúdo dos campos de rótulo, o resultado da operação é transferido para o registrador destino (cujo bit de ocupação passa a valer '0') e para as *reservation stations* que estiverem esperando por esse resultado. No nosso exemplo, a partir desse momento, a primeira instrução de adição poderá ser iniciada.

A Figura 2.2 apresenta o estado dos campos de rótulo (*Rot*) de algumas *reservation stations* e registradores e de seus bits de ocupação (*Bit*), após os eventos relacionados com a execução das 2 primeiras instruções do fragmento de programa apresentado.

2.3 Tratamento de Dependências Falsas

O algoritmo de Tomasulo continua o processo de escalonamento mesmo que existam os outros dois tipos de dependências (dependência de saída e anti-dependência). Ao copiar a identificação da *reservation station* que armazena a instrução responsável pela avaliação do novo valor de um registrador para o seu campo de rótulo, o algoritmo garante que as instruções subseqüentes que tenham este registrador como operando irão utilizar o valor correto.

Dizemos que existe uma dependência de saída quando um mesmo registrador é especificado como destino por duas instruções. Para ilustrar o comportamento do algoritmo quando em presença de dependências de saída entre instruções, consideremos o seguinte fragmento de programa:

$$R_0 \leftarrow R_0 \times R_1$$

$$R_0 \leftarrow R_1 + R_2$$

Como o registrador R_0 é especificado como operando destino por ambas as instruções, dizemos que existe uma dependência de saída entre elas.

Vamos assumir que, inicialmente, todas as *reservation stations* estão livres e todos os registradores são válidos. A instrução de multiplicação é despachada para a *reservation station MD1*. Como os bits de ocupação dos operandos fonte (R_0 e R_1) indicam que tais registradores são válidos, o conteúdo deles é transferido para *MD1* durante o despacho. Para indicar que a instrução armazenada em *MD1* irá alterar o conteúdo de R_0, o campo de rótulo deste registrador recebe a identificação de *MD1* e o seu bit de ocupação passa para '1'.

A instrução de adição é despachada para a *reservation station S1*. Por serem válidos (bit de ocupação igual a '0') os conteúdos dos registradore que atuam como operando fonte (R_1 e R_2) são transferidos para *S1*. O campo de rótulo do operando destino (R_0) é atualizado com a identificação de *S1* para indicar que a instrução desta *reservation station* irá alterar o conteúdo de R_0. A identificação

armazenada anteriormente no campo de rótulo associado a R_0 (*MD1*) é destruída. O bit de ocupação, por sua vez, mantém o mesmo valor ('1').

Verificamos que durante o processo de escalonamento, o campo de rótulo dos registradores é continuamente atualizado (com a identificação da *reservation station* cuja instrução especifica esse registrador como destino da operação). A qualquer instante, o conteúdo desse campo de rótulo identifica a última instrução que especificou o registrador correspondente como destino. Em outras palavras, se diversas instruções (já despachadas) especificarem como destino um mesmo registrador, somente a última dessas instruções é que irá alterar o conteúdo desse registrador.

No nosso exemplo, ao término da execução da multiplicação, o resultado da operação e a identificação de *MD1* serão propagados através do *CDB*. Porém, como o conteúdo do campo de rótulo de R_0 difere da identificação propagada pelo *CDB*, o resultado não será copiado para esse registrador. Quando a execução da instrução de adição chegar ao fim, o resultado propagado através do *CDB* será copiado para R_0, pois o campo de rótulo deste registrador armazena a identificação de *S1* (a menos que tenha sido despachada uma instrução posterior a adição que especifique R_0 como operando destino). O bit de ocupação correspondente passa a valer '0'.

Ainda que, devido aos tempos de latência das instruções, a execução da operação de multiplicação termine após a da operação de adição, somente esta poderá atualizar R_0. Neste caso, quando o resultado da multiplicação for propagado, o valor do bit de ocupação de R_0 será '0', conforme a atualização feita ao término da execução da adição. Uma vez que o bit de ocupação estará indicando que o registrador não espera por atualizações, tal resultado não será copiado para R_0.

Portanto, após a execução de um conjunto de instruções com dependências de saída, o valor armazenado no registrador destino independe da ordem de término das instruções.

As Figuras 2.3 e 2.4 apresentam os estados dos dispositivos funcionais após os eventos relacionados com a execução das instruções do fragmento de

Eventos	*Reservation Stations*				*Registradores*					
	MD1		*S1*		R_0		R_1		R_2	
	Rot	*Rot*	*Rot*	*Rot*	*Bit*	*Rot*	*Bit*	*Rot*	*Bit*	*Rot*
Despacho Multiplicação	-	-	-	-	1	*MD1*	0	-	0	-
Despacho Adição	-	-	-	-	1	*S1*	0	-	0	-
Término Adição	-	-	-	-	0	-	0	-	0	-
Término Multiplicação	-	-	-	-	0	-	0	-	0	-

Figura 2.3: Dependências de Saída (Adição terminando antes da Multiplicação)

Eventos	*Reservation Stations*				*Registradores*					
	MD1		*S1*		R_0		R_1		R_2	
	Rot	*Rot*	*Rot*	*Rot*	*Bit*	*Rot*	*Bit*	*Rot*	*Bit*	*Rot*
Despacho Multiplicação	-	-	-	-	1	*MD1*	0	-	0	-
Despacho Adição	-	-	-	-	1	*S1*	0	-	0	-
Término Multiplicação	-	-	-	-	1	*S1*	0	-	0	-
Término Adição	-	-	-	-	0	-	0	-	0	-

Figura 2.4: Dependências de Saída (Multiplicação terminando antes da Adição)

programa apresentado. Na Figura 2.3 está representada a situação em que o término da execução da instrução de adição ocorre antes do término da multiplicação. Já a Figura 2.4 representa a situação em que o término da execução da instrução de adição ocorre após o término da multiplicação.

Quando o registrador destino de uma instrução é utilizado como operando fonte por uma anterior dizemos que elas são anti-dependentes. Utilizaremos o fragmento de programa a seguir, para ilustrar o comportamento do algoritmo quando na presença de anti-dependências:

$$R_2 \leftarrow R_0 \times R_1$$

$$R_0 \leftarrow R_1 + R_3$$

Como R_0 é operando fonte da multiplicação e destino da adição, estas instruções são anti-dependentes. Dependendo do estado do bit de ocupação de R_0 quando a primeira dessas instruções for escalonada, diferentes ações são tomadas. Se o bit de ocupação indicar que o conteúdo é válido, o conteúdo do registrador é transferido para a *reservation station* no momento do escalonamento da instrução.

Por outro lado, se o bit de ocupação indicar que o conteúdo de R_0 não é válido, então a multiplicação possui uma dependência verdadeira de dados

com uma instrução anterior. Esta instrução poderia ser, por exemplo:

$$R_0 \leftarrow R_1 + R_4$$

Como vimos na seção anterior, neste caso o conteúdo do campo de rótulo de R_0 é transferido para a *reservation station* no momento em que a multiplicação for despachada. O valor do operando fonte representado por R_0 será copiado do *CDB* pela *reservation station* que armazena a multiplicação, quando do término da primeira instrução de adição. Assim, qualquer que seja a situação, a instrução fica imune a alterações no valor deste registrador por instruções posteriores.

Com a utilização deste esquema de rotulação, técnicas alternativas para o tratamento de falsas dependências, como por exemplo a renomeação de registradores, tornam-se desnecessárias.

Capítulo 3

Modelos de Simulação

Durante a realização deste trabalho diversos modelos de máquina foram especificados, simulados e avaliados. Através dos experimentos levados a cabo com esses modelos, foi possível observar melhor o funcionamento do algoritmo de Tomasulo, e avaliar a importância dos componentes (*reservation stations e CDB*) e das atividades (despacho e rotulação) por ele realizadas. Uma vez reconhecida a importância desses dispositivos e atividades, eles foram incorporados definitivamente nos demais modelos. O sistema de memória, nos modelos iniciais, limitava o desempenho da máquina e por esse motivo, decidimos empregar um esquema alternativo nos outros sistemas. Outros mecanismos propostos foram posteriormente incluídos e avaliados. Assim, passo a passo evoluiu-se até o modelo final de máquina Super Escalar. Estes modelos são descritos a seguir.

3.1 A Máquina Básica

Na modelagem de uma máquina Super Escalar é essencial determinar o conjunto de instruções e a mistura adequada de unidades funcionais. Tendo em vista que estávamos interessados em examinar o efeito do escalonamento dinâmico no desempenho de máquinas Super Escalares, decidimos então incluir no nosso modelo as características de uma arquitetura Super Escalar comercialmente disponível. Por essa razão é que foram incorporados no nosso modelo, o repertório de instruções e os tipos de unidades funcionais do processador i860 da Intel [INTE89].

Dentre as instruções do i860 somente as que são executadas no modo dual e as *pipelined* não foram incorporadas. As instruções do tipo dual são utilizadas em processadores cujos algoritmos de detecção de paralelismo são implementados no compilador (escalonamento estático). Como no algoritmo de Tomasulo esta tarefa é realizada pelo *hardware*, o modo dual foi ignorado. Conforme apontado por Pleszkun [PLES88] a utilização de técnicas de *pipeline* nas unidades funcionais de um processador não vetorial, não proporciona uma melhora superior a 4 % no desempenho. Por esse motivo as unidades funcionais do nosso modelo não utilizam tal tecnologia.

O conjunto de instruções do i860 opera sobre um banco de registradores inteiros e um banco de ponto flutuante. Nos nossos modelos, esses dois bancos de registradores possuem a mesma configuração como no i860. Os dados e as instruções são armazenadas em sistemas de memória que variam de modelo para modelo.

Excetuando a unidade *Core*, a funcionalidade das outras unidades do i860 foi mantida no nosso modelo. A unidade *Core* deixou de ser responsável pela execução das instruções de *Load/Store*. Para realizar estas operações, introduzimos um outro tipo de dispositivo funcional: a unidade de Acesso à Memória.

Desse modo, no nosso modelo de arquitetura, as seguintes funções são realizadas por cada tipo de unidade:

Core - Responsável pela execução de todas as operações lógicas e aritméticas com inteiros, instruções de desvio e de transferência de dados de um registrador inteiro para um outro de ponto flutuante.

Soma - Esta unidade executa as instruções de soma, subtração e comparação com reais, bem como a instrução de conversão de formato de ponto flutuante para inteiro.

Multiplicação - A unidade de multiplicação realiza as operações de multiplicação com reais e a avaliação do recíproco de um real (usada para implementar a divisão de reais).

Gráfica - Ela executa operações aritméticas sobre valores de tipo inteiro longo armazenados no banco de registradores de ponto flutuante. Executa, também, instruções para facilitar a implementação de algoritmos gráficos tridimensionias. Dentre estes algoritmos encontram-se o de sombreamento de *pixels* e o *Z-buffer* para a eliminação de superfícies encobertas. A operação de transferência de dados de registradores de ponto flutuante para os inteiros também é realizada por esta unidade.

Acesso à Memória - As instruções de transferência de dados entre o sistema de memória e os bancos de registradores, são executadas por este tipo de unidade funcional.

3.2 Modelos Prévios

Os primeiros experimentos levados a cabo estão descritos em [BAR92a]. No modelo utilizado, associamos a cada grupo de unidades funcionais de um determinado tipo da máquina básica, um conjunto de *reservation stations*. Graças a rede de conexão entre esses dois conjuntos de dispositivos, todas as unidades funcionais de um mesmo tipo tinham acesso a qualquer uma das *reservation stations* pertencentes ao conjunto associado à elas.

Para implementar o esquema de rotulação descrito no Capítulo 2, acrescentamos um campo de rótulo e um bit de ocupação nos registradores dos dois bancos. Além disso, foi incluído um *CDB* para propagar os resultados produzidos pelas unidades funcionais, para as *reservation stations* e registradores.

Nesses modelos, as instruções eram buscadas, decodificadas e escalonadas seqüencialmente. Assim, a busca de uma instrução só seria iniciada após o escalonamento da anterior. Como conseqüência da utilização do algoritmo de Tomasulo em todos os tipos de unidade funcional, o mecanismo de escalonamento de instruções foi alterado para tratar mais uma condição de parada. Além da inexistência de *reservation stations* disponíveis, o escalonamento de uma instrução de desvio condicional passou a ser uma condição de interrupção do mecanismo de des-

pacho.

Somente quando todos os dados necessários para a execução de uma instrução estiverem disponíveis na *reservation station* e quando uma unidade funcional estiver livre, é que a instrução pode ser iniciada. Havendo mais instruções prontas para serem executadas do que unidades funcionais disponíveis, a prioridade no atendimento é para aquelas que foram escalonadas há mais tempo. O algoritmo permite que uma instrução com todos os seus operandos disponíveis, seja executada antes de outra escalonada há mais tempo mas que não esteja pronta para ser executada. A única exceção é para a unidade de Acesso à Memória. Nessa unidade, as *reservation stations* são organizadas segundo uma fila. Assim, uma instrução só é executada quando todas as demais instruções de acesso à memória escalonadas anteriormente já tiverem começado a ser executadas. Através desse esquema garante-se a consistência dos dados na memória.

No nosso modelo, a memória é organizada em duas partições, uma para o armazenamento de instruções e outra para dados. Foi assumida a existência de um sistema de acesso a instruções extremamente rápido, como uma memória *cache*, por exemplo. Contudo, omitimos a descrição do comportamento desse tipo de memória. Assumimos que a busca de qualquer instrução levaria um mesmo intervalo de tempo t relativamente pequeno.

A memória de dados ficou organizada sob a forma de um ou mais bancos, cada um deles associado à uma unidade funcional de acesso à memória, sendo esta a única capaz de acessá-lo. Os bancos de memória são numerados em ordem crescente e os endereços de memória foram distribuídos através dos bancos com cada grupo de quatro *bytes* consecutivos armazenados num mesmo banco. Desta forma, é possível realizar simultaneamente tantos acessos à memória quantos forem os bancos.

Ao término da execução de uma instrução, é feito um pedido de acesso ao *CDB* para propagar o resultado da operação. Se mais de uma unidade requisitar o *CDB* simultaneamente, a prioridade no uso é para a unidade que começou a executar a sua instrução há mais tempo.

Nos modelos que foram descritos posteriormente [FER92a,FER92b], foram introduzidas modificações para aprimorar o modelo inicial e atingir um desempenho maior.

O sistema de memória de instruções foi alterado de modo que o comportamento de uma memória *cache* com as características da existente no i860 fosse reproduzido. Introduzimos intervalos de tempo distintos para o acesso a esse sistema caso ocorra um acerto ou uma falha.

Com o objetivo de investigar a contribuição (no desempenho do modelo) oferecida pelo *CDB* e pela presença de *reservation stations*, configurações incorporando (ou não) esses componentes foram especificadas. Desse modo, obtivemos as seguintes variações de arquiteturas derivadas do modelo básico:

a) sem *CDB* e sem *reservation stations*: Como não existe nenhuma *reservation station*, as instruções são transferidas diretamente para um *buffer* nas unidades funcionais, capaz de armazenar, no máximo, uma instrução. Como não existe *CDB*, os resultados gerados são transferidos para os registradores de onde são lidos, como operandos, pelas unidades funcionais.

b) com *CDB* e sem *reservation stations*: Essa variação possui um *CDB* conectando as unidades funcionais e os bancos de registradores da máquina. Através do *CDB*, os resultados produzidos pelas unidades funcionais são transferidos diretamente para um dos bancos de registradores.

c) sem *CDB* e com *reservation stations*: Associado a cada conjunto de unidades funcionais de um determinado tipo existe pelo menos uma *reservation station.* O código de operação e os operandos de cada instrução são transferidos para uma *reservation station* antes dela ser executada. Como não há *CDB*, as *reservation stations* buscam os operandos no banco de registradores.

d) com *CDB* e com *reservation stations*: Nesta variação, os dados relativos a cada instrução são transferidos para as *reservation stations.* Os resultados gerados pelas unidades funcionais podem ser transferidos para o banco de registradores e para as *reservation stations* simultaneamente.

Nas variações (a) e (c) (modelos sem *CDB*), um esquema do tipo *scoreboard* [THOR64] é responsável pelo tratamento das dependências. A inexistência, nesses modelos, de um *CDB* (e do correspondente esquema de propagação de rótulos e de valores) provoca uma condição adicional de interrupção no processo de escalonamento de instruções: toda vez que o registrador destino de uma instrução a ser escalonada já tiver sido designado para armazenar o resultado de uma instrução anterior, o processo é interrompido.

Para garantir a consistência dos dados armazenados na memória, empregamos a política *FCFS* (*First Come First Served*) na execução das instruções de *Load/Store*. Assim, nas variações (a) e (b) (modelos sem *reservation stations*), para que não tivéssemos uma condição adicional de interrupção no processo de escalonamento de instruções, foi permitido que as unidades funcionais de acesso à memória referenciassem qualquer um dos bancos de memória. Se cada unidade de acesso à memória estivesse ligada a somente um banco, o processo de escalonamento seria interrompido sempre que a unidade funcional ligada ao banco de memória que contém o endereço apontado pela instrução de *Load/Store* que se pretende despachar, estivesse ocupada.

3.3 Modelo Final

O modelo descrito a seguir foi o utilizado para avaliar o efeito do escalonamento dinâmico em processadores Super Escalares. Esse modelo é resultante da eliminação das principais deficiências que limitavam o desempenho dos modelos prévios. Adicionalmente, o modelo incorpora algumas técnicas modernas utilizadas em processadores atuais.

Nas configurações anteriores da máquina básica, o tempo de latência requerido pelas operações de *load* e *store* era relativamente longo quando comparado com o tempo de latência das instruções executadas pela *Core*. Por esse motivo, a memória se tornava um gargalo que penalizava o desempenho do processador, além de dificultar a avaliação do efeito dos demais dispositivos do processador e do seu algoritmo de escalonamento dinâmico. Ao introduzirmos um outro nível hierárquico

no subsistema de memória, nosso modelo ficou mais semelhante aos processadores Super Escalares atuais.

A capacidade de armazenamento da memória *cache* que foi incorporada no nosso modelo é igual a do processador i860, e a memória principal ficou organizada segundo um único banco.

O acesso ao sistema de memória de dados passou a ser feito através de uma única unidade funcional. Dessa forma, duas ou mais instruções de acesso à memória não podem ser executadas simultaneamente. Associada a esta única unidade funcional, um número variado de *reservation stations* ficam organizadas segundo uma fila. As instruções despachadas para estas *reservation stations* são enfileiradas de acordo com a ordem de escalonamento. Este é o critério usado para determinar a ordem de execução dessas instruções. Numa das variações desse modelo, decidimos incluir um método de remoção de ambigüidades [BAR92b]. Através desse método, a ordem de atendimento das operações de *load* e *store*, pode ser relaxada em algumas situações, deixando de obedecer ao critério *FCFS* (*First Come First Served*). Dessa forma, operações com a memória podem ser antecipadas nas seguintes situações:

1. um *Load* pode ser antecipado quando não houver nenhuma operação de *Store*, anterior a ela na fila, que acesse as mesmas posições de memória.

2. um *Store* pode ser antecipado quando não houver nenhuma operação de *Load* ou *Store*, anterior a ela na fila, referenciando a mesma posição de memória.

Portanto, para que uma instrução de *Load* seja antecipada, é necessário que os endereços de todas as instruções de *Store* anteriores a ela já tenham sido avaliados. Já as instruções de *Store* exigem também, a avaliação do endereço das operações de *Load* anteriores a ela. Caso exista mais de uma instrução capaz de ser antecipada será dada prioridade aquela despachada a mais tempo.

O algoritmo de escalonamento de instruções foi modificado de modo que múltiplas instruções pudessem ser buscadas, decodificadas e despachadas simultaneamente [BAR92b]. Baseado no trabalho [DWYE87], assumiu-se que o tempo

gasto na realização destas tarefas para múltiplas instruções simultaneamente, seria o mesmo que para uma instrução de cada vez. Além do mais, estávamos interessados em avaliar a eficiência desta política de escalonamento levando em consideração somente as limitações impostas pelo algoritmo de Tomasulo e pelas características dos componentes do *benchmark*.

Neste modelo, o "número n de *CDB's*" do processador passou a ser um parâmetro de simulação. Assim, tornou-se possível propagar os resultados de até n instruções que terminassem simultaneamente.

Também foi incluído um mecanismo de predição de desvios condicionais. Este método não está baseado em nenhuma proposta realista conhecida. Durante a simulação, ao se escalonar uma instrução de desvio condicional, é gerado um número aleatório que determinará se a predição deste desvio será correta ou não. A chance de acerto é passada como parâmetro de simulação através de um número denominado "taxa de acerto." O motivo que nos levou a esta opção, foi a nossa preocupação em determinar um limite máximo teórico para o desempenho deste modelo ao utilizar algum método realista de predição de desvios. Este limite pode ser avaliado atribuindo-se o valor 100% à taxa de acerto.

Os tempos de latência de cada unidade funcional são expressos em termos de ciclos de máquina. As seguintes latências foram usadas durante nossos experimentos:

- decodificação e escalonamento de instruções: um ciclo cada;
- busca de instruções: um ciclo para *hit* ou dezesseis para *miss* na *cache* respectivamente;
- operações na unidade de adição de ponto flutuante: seis ciclos;
- operações na unidade de multiplicação de ponto flutuante: seis ou nove ciclos, para simples e dupla precisão respectivamente;
- operações de acesso a *cache* de dados: três ou dezoito ciclos, para *hit* ou *miss* respectivamente;
- operações na unidade funcional *core*: três ciclos;
- operações na unidade gráfica: seis ciclos

Assumiu-se que a busca de operandos para as instruções no banco de registradores seria feita durante a decodificação, que abrangeria as seguintes fases:

- decodificação;
- busca de operandos

Para garantir o funcionamento correto do mecanismo de despacho associativo, os resultados propagados pelo *CDB* não podem alterar o banco de registradores durante a fase de busca de operandos. Portanto, quando resultados forem propagados durante o ciclo de decodificação, o banco de registradores será atualizado durante a primeira fase da decodificação.

O escalonamento de instruções também foi dividido em duas fases:

- despacho;
- avaliação do *CDB*

Durante o despacho, os parâmetros para a execução das instruções (obtidos durante a decodificação) são transferidos para as *reservation stations* disponíveis. Na segunda fase do escalonamento, as *reservation stations* verificam se o *CDB* está propagando algum operando esperado por elas. Caso afirmativo, a *reservation station* armazena este operando no campo apropriado. Desta forma evitamos que instruções despachadas no mesmo ciclo em que algum de seus operandos foi propagado, fiquem esperando eternamente tais operandos.

Capítulo 4

Método de Avaliação

Este capítulo apresenta uma breve descrição do nosso meio ambiente de avaliação e da metodologia utilizada na condução de nossos experimentos. A bateria de programas de teste, a máquina referência e as características do nosso processo de simulação de Arquiteturas Super Escalares, serão discutidos a seguir.

4.1 A Máquina Referência

Para avaliar o modelo de máquina Super Escalar descrito no Capítulo 3, comparamos o seu desempenho com o de uma outra que servisse como referência. No modelo utilizado como referência as instruções são executadas seqüencialmente. Uma instrução só é buscada após a conclusão da anterior. Os tempos de latência das unidades funcionais da máquina Super Escalar foram mantidos na máquina referência. Comparando os desempenhos dos dois modelos, podemos verificar as vantagens oferecidas pelas técnicas de extração do paralelismo de baixo nível que foram exploradas pelos nossos experimentos.

4.2 Programas Teste

Para avaliar o efeito do escalonamento dinâmico no desempenho de máquinas super escalares, submetemos o nosso modelo de máquina super escalar a um conjunto de situações freqüentemente encontradas na prática. Assim, foi selecionado um

conjunto de oito programas, formando a bateria de teste (*benchmark*), com diferentes características. Os programas da bateria de teste são:

- Integral
- Decomposição LU
- Hu-Tucker
- Branch
- BCDBin
- Livermore Loop 24
- Quick-Sort
- Bubble-Sort

O programa **Integral** (Int) realiza o cálculo aproximado, segundo o método dos trapézios [CONT65], da integral da função $f(x) = x^2$ para $0 \leq x < 30$ e altura do trapézio igual a 0,1; o de **Decomposição LU** (LU) [FORS67] faz a decomposição de uma matriz de dimensão 10 x 10; o **Hu-Tucker** (Hu) [YOHE72] implementa um algoritmo de codificação binária alfabética com redundância mínima; o programa **Branch** (Brc) [DITZ87] realiza a contagem do total de números pares e ímpares contidos no intervalo $[0, 2559]$ e calcula o somatório de tais números; o **BCDBin** (Bcd) [AUHT85] realiza a conversão para formato binário de 40 números codificados em *BCD*; o **Livermore Loop 24** (Liv) [MCMA83] determina o índice do menor componente de um vetor de dimensão 700; o programa **Quick-Sort** (Qck) [KNUT73] classifica um vetor de 200 posições; finalmente, o programa **Bubble-Sort** [KNUT73] realiza a ordenação de um vetor de dimensão 35.

Codificados em linguagem C, os programas da bateria foram traduzidos para o código objeto do i860 por um compilador comercial. A Tabela 4.1 mostra o número de ciclos (na máquina referência) requerido por cada programa, o número de instruções do i860 executadas por cada tipo de unidade funcional, e o total de instruções executadas.

Examinando essa tabela, podemos constatar que somente os códigos dos programas Integral e Decomposição LU possuem instruções que são executadas pelas unidades funcionais de Multiplicação, Adição e Gráfica. Além disso, a razão entre a quantidade de instruções destes tipos e o total geral de instruções executadas, é bem maior no programa Integral do que no LU. O programa Branch, por sua vez, é o único componente da bateria cujas variáveis estão todas armazenadas em

Programa	Core	Mult	Ad	Mem	Gra	Total	Ciclos
Integral	8.083	1.196	2.991	2.995	300	15.565	109.410
LU	10.406	555	506	4.111	33	15.611	97.781
Hu-Tucker	9.544	0	0	4.605	0	14.149	86.399
Branch	19.206	0	0	0	0	19.206	116.563
BCDBin	16.184	0	0	2.501	0	18.649	113.142
Livermore	11.189	0	0	4.369	0	15.558	94.932
Quick-Sort	16.995	0	0	4.268	0	21.263	129.671
Bubble-Sort	10.282	0	0	5.808	0	16.090	97.314

Tabela 4.1: Total de Instruções por Unidade Funcional e Ciclos

registradores. Por esse motivo nenhuma instrução foi executada pela unidade de Acesso à Memória.

4.3 A Simulação

Uma vez traduzidos para o código objeto os programas de teste foram processados por um simulador do i860 e os correspondentes arquivos contendo os *traces* da execução foram produzidos. Cada arquivo de *trace* é composto por um conjunto de índices, cada um associado a uma instrução do arquivo de código seqüencial. Esses dois índices correspondem à sequência de execução destas instruções. Desta forma, evitou-se que as informações relativas à uma instrução fossem repetidas desnecessariamente, como aconteceria se, em vez de índices, utilizássemos o código das instruções no *trace*.

Além da simulação, os arquivos contendo o código objeto do i860 passam por uma etapa de pré-processamento. Durante essa etapa, é gerado um arquivo, denominado arquivo de código seqüencial que armazena informações relacionadas com as instruções do programa.

O comportamento da máquina Super Escalar, ao executar cada um dos programas da bateria de teste, é reproduzido levando em consideração apenas as dependências de dados existentes entre as instruções do arquivo de código sequencial quando despachadas na ordem indicada pelo *trace*, e de acordo com os tempos de latência. Durante a simulação destes programas no nosso modelo de máquina

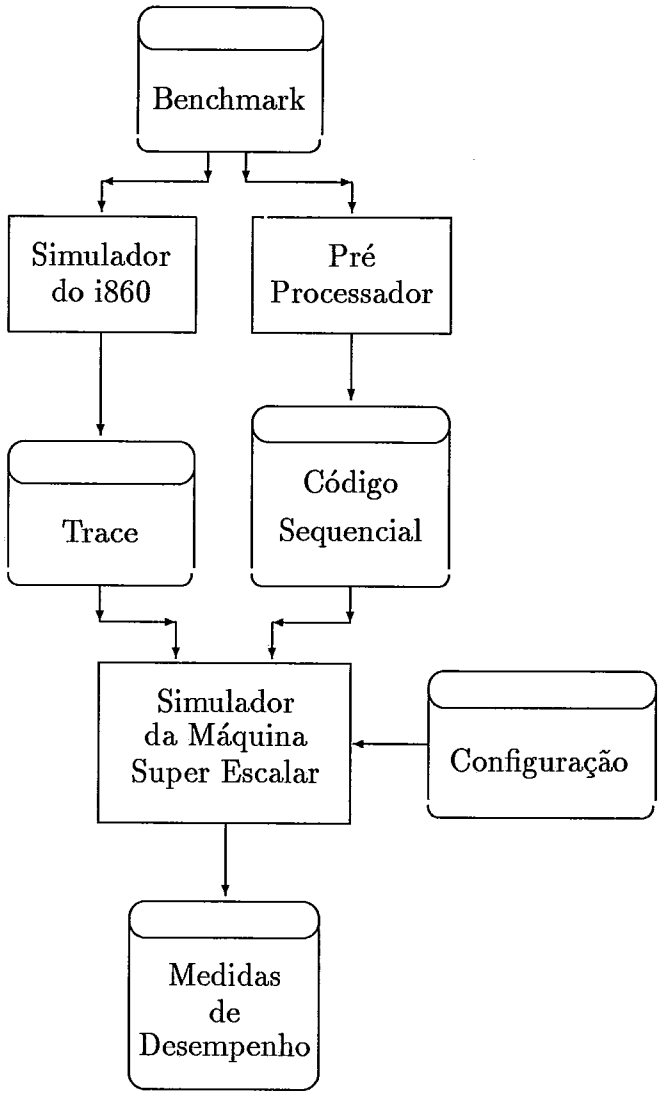

Figura 4.1: Etapas do Processo de Simulação

Super Escalar, as instruções não são efetivamente executadas. Esta execução ocorre somente uma vez para cada programa da bateria de teste e é realizada pelo simulador do i860.

Além do *trace*, o simulador lê um "arquivo de configuração" contendo os seguintes parâmetros:

- Total de *reservation stations* associadas a cada tipo de unidade funcional
- Total de unidades funcionais de cada tipo
- Número máximo de instruções que podem ser escalonadas simultaneamente (Tamanho da Janela de Instruções)
- Total de CDB's
- Taxa de acerto do método de predição de desvios

Ao término da simulação de um programa na máquina Super Escalar, um relatório é gerado. Os dados apresentados neste relatório incluem:

- Total de ciclos requeridos pelo programa na máquina Super Escalar
- Total de ciclos requeridos na máquina referência
- Taxa de aceleração ($speed - up$)
- Taxa de redução
- Tempo de ocupação de cada *reservation station*
- Tempo que um certo número de *reservation stations* permaneceram ocupadas
- Tempo de espera por operandos de cada *reservation station*
- Tempo de ocupação de cada unidade funcional
- Tempo em que um certo número de unidades funcionais ficaram ocupadas
- Tempo que um certo número de CDB's ficaram ocupados

A Figura 4.1 ilustra as diversas etapas do nosso modelo de simulação.

Realizando várias simulações com diferentes parâmetros e avaliando os resultados gerados no relatório, foi possível avaliar o efeito do escalonamento dinâmico no desempenho de máquinas super escalares.

Capítulo 5

Resultados

O balanceamento de um processador Super Escalar constitui-se num dos mais sofisticados problemas que o arquiteto precisa enfrentar. Incorporando um grande número de diferentes tipos de recursos, garante-se que o processamento em paralelo de múltiplas instruções não será interrompido pela contenção no uso das facilidades do *hardware* subjacente.

Usando como critério o compromisso "custo × desempenho," podemos verificar que essa solução é bastante simplista: durante o processamento dos programas de aplicação, somente uma pequena fração dos recursos é que estará sendo utilizada.

Por outro lado, se reduzirmos —indiscriminadamente— os recursos do *hardware*, podemos provocar um desbalanceamento no processador: dependendo do programa de aplicação, alguns tipos de recursos podem apresentar uma elevada taxa de utilização, enquanto outros permanecerão ociosos durante o processamento.

Ao longo dos nossos estudos, desenvolvemos uma metodologia para projetar arquiteturas Super Escalares balanceadas. Reproduzindo o comportamento de diversos algoritmos de escalonamento de instruções, essa metologia foi utilizada na condução dos experimentos descritos em [BARB92a], [BARB92b], [FERN92a] e [FERN92b].

Nesse capítulo apresentaremos os resultados obtidos através do emprego dessa metodologia na condução de um novo conjunto de experimentos. Essa

nova série de experimentos diferencia-se da anterior pela capacidade do algoritmo de escalonamento que foi implementado. Em vez de uma única instrução por ciclo de máquina, o novo algoritmo é capaz de despachar múltiplas instruções simultaneamente.

Através dos experimentos realizados com esse algoritmo de despacho, foi possível avaliar o impacto de importantes parâmetros arquiteturais no desempenho do nosso modelo de máquina Super Escalar.

Exemplos típicos de parâmetros que foram investigados incluem: o número de instruções escalonadas simultaneamente (i.e., o tamanho da janela); o número e a mistura de dispositivos funcionais e *reservation stations* que devem ser introduzidos no processador, de modo que tenhamos uma arquitetura balanceada; qual o número ideal de CDBs que devemos utilizar; qual o efeito na taxa de ocupação dos recursos do processador quando modificamos o total e a mistura de unidades funcionais e *reservation stations*; o que realmente ocorre no interior do processador Super Escalar se um mecanismo de predição de desvios for incorporado ou não.

5.1 Impacto do Tamanho da Janela

Conforme apontado por diversos autores (vide [ACOS86,THOR64]), despachar somente uma instrução por ciclo de máquina reduz a taxa de utilização dos recursos funcionais. Para avaliar o efeito produzido pelo despacho de múltiplas instruções por ciclo, modificamos, no nosso modelo de máquina Super Escalar, o algoritmo de Tomasulo. Desse modo, ao invés de uma única instrução por ciclo (como ocorre nas unidades de ponto flutuante do IBM 360/91), nosso algoritmo alternativo é capaz de despachar diversas instruções simultaneamente.

Essas instruções são transferidas da memória *cache* (no interior do processador) ou da memória principal para uma área de armazenamento denominada "janela de instruções." Durante nossos experimentos, empregamos janelas capazes de armazenar 1, 2, 4, 8 e 16 instruções. A cada 3 ciclos de processador, o algoritmo de escalonamento examina as instruções contidas na janela, despachando-as para as

reservation stations apropriadas. Após terem sido despachadas, as instruções são removidas da janela, e instruções subseqüentes irão preencher a janela.

A indisponibilidade de *reservation stations* provoca a interrupção no processo de despacho. Nesse caso, as instruções precedendo aquela que provocou a interrupção, são despachadas e removidas da janela normalmente. As instruções restantes são movimentadas para o início da janela (preenchendo as posições ocupadas pelas instruções que foram despachadas), e instruções subseqüentes serão trazidas para preencher o restante da janela. Tão logo ocorra a liberação de uma *reservation station* apropriada, o processo de despacho é retomado.

Procedimento análogo ocorre para instruções de desvio condicional. Após o despacho e remoção das instruções da janela que precedem um desvio condicional, o processo é interrompido, assim permanecendo até que a condição do desvio esteja avaliada. No caso do desvio ser tomado, o algoritmo fica aguardando até que o endereço efetivo alvejado esteja avaliado.

A capacidade da janela de instruções é um parâmetro de simulação, e variando esse parâmetro, simulamos a execução dos programas de teste no nosso modelo de máquina Super Escalar. Nesses experimentos, configuramos a máquina alvo com:

$$\begin{cases} 64 & \textit{CDBs}; \\ 64 & \textit{reservation stations}; \\ 64 & \text{unidades funcionais de cada tipo.} \end{cases}$$

Levando em conta que esses recursos não foram completamente utilizados durante a interpretação dos programas da bateria de teste, podemos concluir que essa configuração pode ser considerada como um processador Super Escalar com recursos ilimitados.

As taxas de aceleração (*speedup ratios*) produzidas pelo despacho de múltiplas instruções, em relação a máquina referência, estão listadas na Tabela 5.1.

A Figura 5.1 mostra o efeito nas taxas de aceleração quando aumentamos o tamanho da janela de instruções. Como podemos observar, o programa *Integral* tem um comportamento especial em relação aos demais: embora a taxa de

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	2,11	3,40	4,91	5,67	6,14
LU	1,95	3,05	3,30	3,31	3,31
Hu-Tucker	1,81	2,65	2,84	2,85	2,85
Branch	1,49	1,86	1,86	1,86	1,86
BCDBin	1,72	2,13	2,45	2,52	2,52
Livermore	1,83	2,29	2,29	2,29	2,29
Quick-Sort	1,72	2,22	2,52	2,59	2,61
Bubble-Sort	1,84	2,51	2,62	2,62	2,62

Tabela 5.1: Variação das Taxas de Aceleração com o Tamanho da Janela

crescimento de sua curva apresente uma queda brusca para janelas com mais de 8 instruções, ele registrou um aumento significativo na taxa de aceleração para a janela com 16 instruções. Esse aumento reflete o volume de paralelismo existente entre as instruções do programa *Integral*, além da pequena parcela de instruções de desvio condicional que foram executadas.

Os programas *Branch* e *Livermore* também apresentam um comportamento particular: a curva da taxa de aceleração de ambos pára de crescer para janelas com mais do que 2 instruções. No caso do programa *Branch*, isto decorre do grande número de instruções de desvio condicional executadas, o que causa a interrupção freqüente do algoritmo de despacho. No caso do programa *Livermore*, este comportamento resulta das inúmeras dependências verdadeiras apresentadas por suas instruções.

Para os outros componentes do *benchmark*, um comportamento mais uniforme foi observado. Embora, em alguns destes casos, tenha ocorrido um crescimento na taxa de aceleração para janelas com até 8 instruções, podemos constatar que aumentos significativos só foram identificados para janelas com 2 e 4 instruções.

Desse modo, podemos afirmar que a taxa de aceleração varia com o tamanho da janela de instruções. Isto ocorre porque nas configurações com janelas de tamanho maior que 1, mais instruções estarão simultaneamente prontas para serem executadas. Contudo, as características de cada programa (e.g., as dependências verdadeiras de dados e os desvios condicionais) limitam o crescimento da taxa de aceleração.

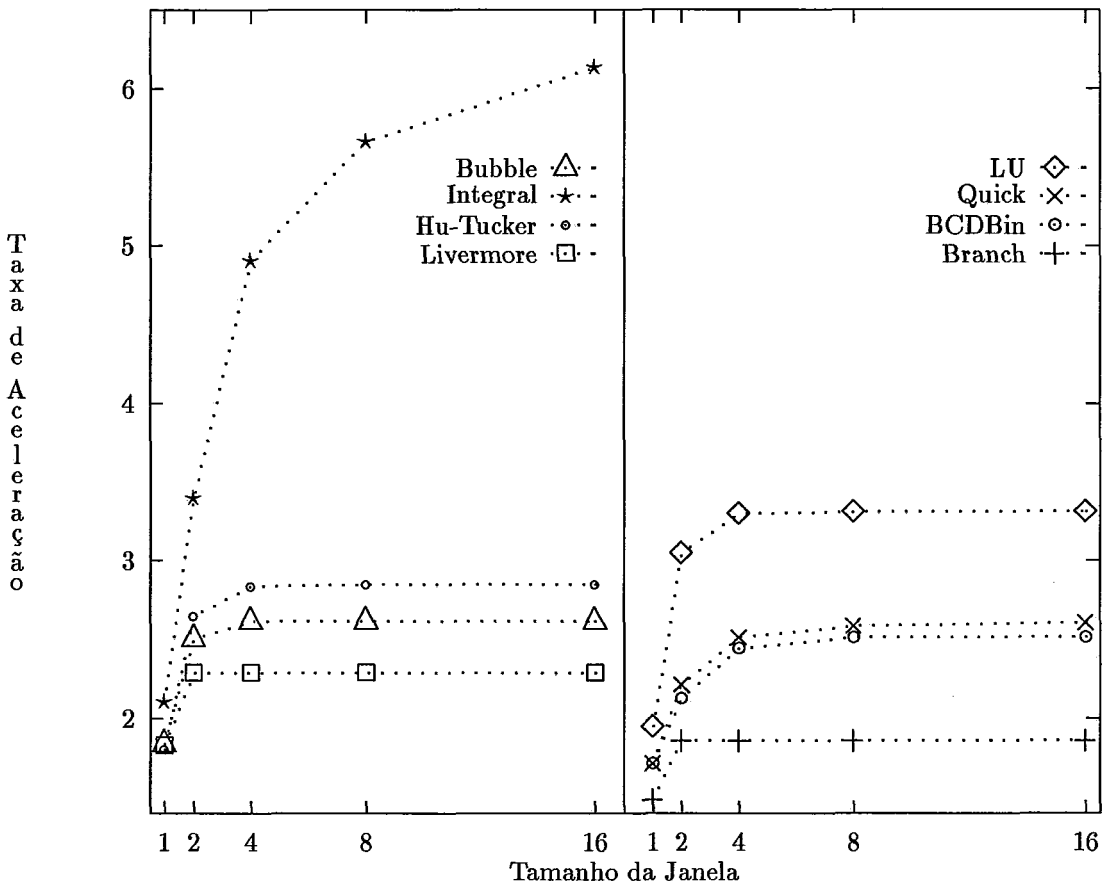

Figura 5.1: Efeito do Despacho Múltiplo de Instruções na Taxa de Aceleração

Embora o despacho múltiplo de instruções propicie uma melhor ocupação dos recursos, nem sempre este objetivo é atingido. Isto pode ser verificado na Figura 5.2 que mostra, para cada programa, a porcentagem do tempo de execução em que todas as unidades funcionais *Core* ficaram ociosas (tempo ocioso) em função do tamanho da janela.

Nos programas *Hu-Tucker*, *Livermore*, *LU* e *Quick-Sort*, a menor porcentagem de tempo ocioso para as unidades *Core* foi atingida para janelas com tamanho 2. Excetuando o programa *LU*, podemos verificar que o nível de ociosidade cresce com o tamanho da janela. No caso do programa *LU*, a taxa de ociosidade estabilizou-se para janelas com mais do que 2 instruções. Para os programas *Integral* e *Bubble*, esta porcentagem decresce até a janela atingir o tamanho 4 e em seguida a porcentagem volta a crescer. Finalmente, os programas *BCDBin* e *Branch* foram os

únicos que apresentaram a menor taxa de ociosidade (das unidades *Cores*) quando a janela armazena 1 única instrução, aumentando progressivamente essa taxa de ociosidade à medida que o tamanho da janela cresce.

Nos programas que apresentam um aumento no nível de ociosidade dos recursos do processador, o crescimento na taxa de aceleração se deve a ocupação simultânea de um maior número de unidades funcionais durante uma pequena fração do tempo de execução, contribuindo desse modo, para uma reduzida utilização dos recursos durante a maior parte do tempo de processamento. Isto pode ser verificado nas Tabelas 5.2, 5.3, 5.4, 5.5 e 5.6 que apresentam, para cada programa teste e para os diversos tamanhos de janela, as porcentagens do tempo de execução em que grupos de unidade *Core* permaneceram ocupadas.

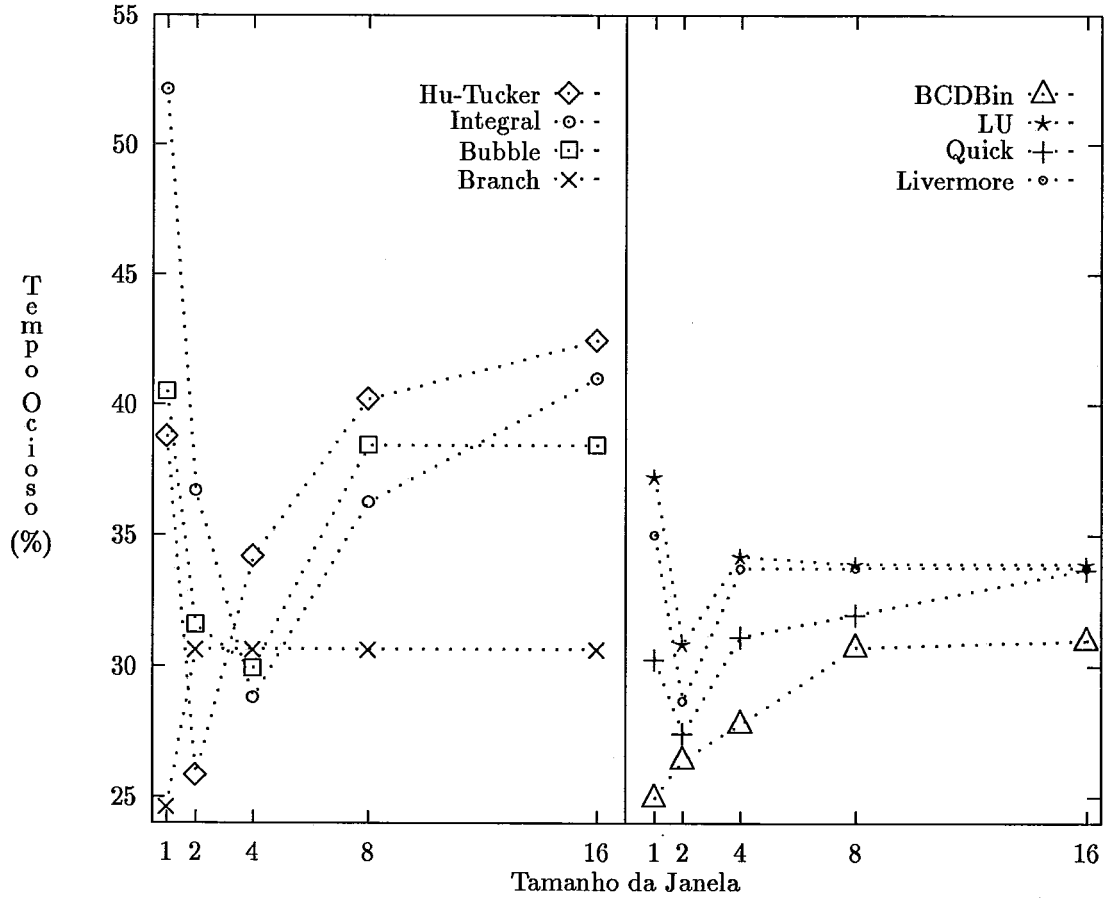

Figura 5.2: Variação do Tempo Ocioso das Unidades *Core*

Examinando os dados apresentados nessas tabelas, verificamos que a

avaliação isolada da taxa de aceleração é insuficiente para que possamos determinar o tamanho ideal da janela que irá assegurar o uso eficiente dos dispositivos do processador. Ignorar o total de recursos utilizados e o nível de ocupação dos mesmos, pode resultar numa configuração de máquina com custo elevado diante dos benefícios obtidos. Para ilustrar essa afirmação, façamos uma análise comparativa de dois programas com diferentes comportamentos: *BCDBin* e *Bubble*. Nesta análise nos preocuparemos somente com os dados obtidos a partir de tamanhos de janela que apresentaram crescimento, ainda que pequeno, na taxa de aceleração.

Programa	0 Core	1 Core	2 Core	3 Core	4 Core	5 Core	> 5 Core
Integral	52,14	47,86	———	———	———	———	———
LU	37,24	62,72	0,04	———	———	———	———
Hu-Tucker	38,77	61,22	0,01	———	———	———	———
Branch	24,63	75,37	———	———	———	———	———
BCDBin	24,95	75,05	———	———	———	———	———
Livermore	35,00	65,00	———	———	———	———	———
Quick-Sort	30,29	69,71	———	———	———	———	———
Bubble-Sort	40,48	59,52	———	———	———	———	———

Tabela 5.2: Ocupação (%) das *Cores* para Janelas de Tamanho 1

Programa	0 Core	1 Core	2 Core	3 Core	4 Core	5 Core	> 5 Core
Integral	36,71	49,34	13,95	———	———	———	———
LU	30,87	43,80	21,75	3,58	———	———	———
Hu-Tucker	25,81	59,59	13,76	0,83	0,01	———	———
Branch	30,66	44,87	24,47	———	———	———	———
BCDBin	26,42	56,90	14,26	2,42	———	———	———
Livermore	28,67	61,19	10,14	———	———	———	———
Quick-Sort	27,44	56,03	15,72	0,81	———	———	———
Bubble-Sort	31,56	55,65	12,79	———	———	———	———

Tabela 5.3: Ocupação (%) das *Cores* para Janelas de Tamanho 2

Programa	0 Core	1 Core	2 Core	3 Core	4 Core	5 Core	> 5 Core
Integral	28,79	43,02	16,12	12,07	———	———	———
LU	34,20	41,02	13,01	8,06	3,71	———	———
Hu-Tucker	34,17	41,57	18,93	4,91	0,22	0,20	———
Branch	30,66	44,87	24,47	———	———	———	———
BCDBin	27,79	50,40	11,15	8,59	1,82	0,25	———
Livermore	33,73	51,07	15,19	0,01	———	———	———
Quick-Sort	31,14	41,92	23,88	0,01	3,05	———	———
Bubble-Sort	29,92	59,93	5,76	4,39	———	———	———

Tabela 5.4: Ocupação (%) das *Cores* para Janelas de Tamanho 4

Programa	0 Core	1 Core	2 Core	3 Core	4 Core	5 Core	> 5 Core
Integral	36,27	26,52	18,63	9,30	9,28	———	———
LU	33,91	37,06	21,37	4,30	3,27	0,09	———
Hu-Tucker	40,19	40,50	8,19	6,95	2,26	1,63	0,28
Branch	30,66	44,87	24,47	———	———	———	———
BCDBin	30,74	47,10	11,20	5,68	3,15	1,87	0,26
Livermore	33,73	56,12	5,08	5,06	0,01	———	———
Quick-Sort	31,97	42,26	16,87	7,54	0,85	0,51	———
Bubble-Sort	38,42	47,32	9,86	0,01	4,38	0,01	———

Tabela 5.5: Ocupação (%) das *Cores* para Janelas de Tamanho 8

Examinando o comportamento do programa *BCDBin*, verificamos que para uma janela com tamanho 1 precisaremos de somente 1 unidade funcional *Core*. Conforme listado na Tabela 5.2, essa unidade permaneceu ocupada durante 75,05 % do tempo de processamento.

Programa	0 Core	1 Core	2 Core	3 Core	4 Core	5 Core	> 5 Core
Integral	40,99	28,73	10,14	5,05	5,04	5,02	5,03
LU	33,91	35,61	21,77	7,84	0,77	0,09	0,01
Hu-Tucker	42,45	39,92	10,16	2,64	1,15	1,28	2,40
Branch	30,66	44,87	24,47	———	———	———	———
BCDBin	31,00	46,84	11,20	5,68	3,15	1,87	0,26
Livermore	33,73	56,12	5,08	5,06	0,01		———
Quick-Sort	33,70	41,29	18,18	2,74	1,02	3,07	———
Bubble-Sort	38,42	51,44	5,74	0,01	0,26	4,13	———

Tabela 5.6: Ocupação (%) das *Cores* para Janelas de Tamanho 16

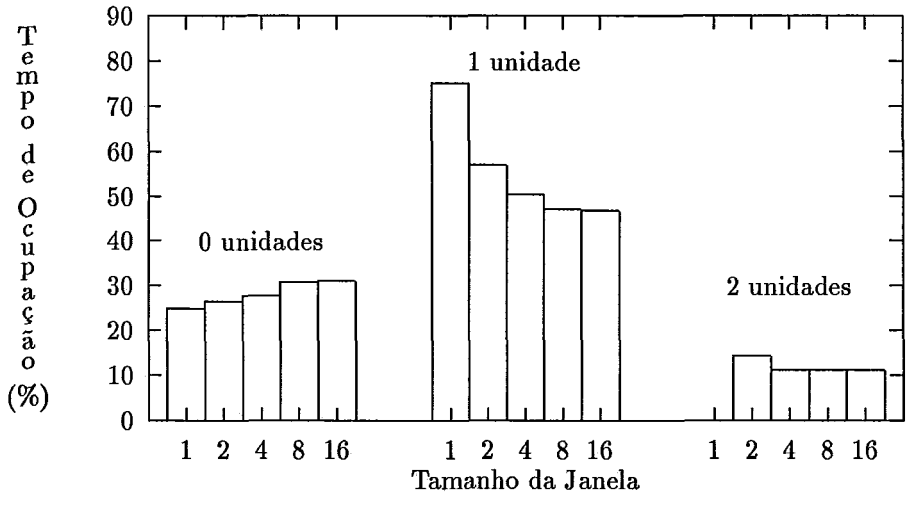

Figura 5.3: Ocupação de Grupos de 0 a 2 Unidades *Cores* no programa *BCDBin*

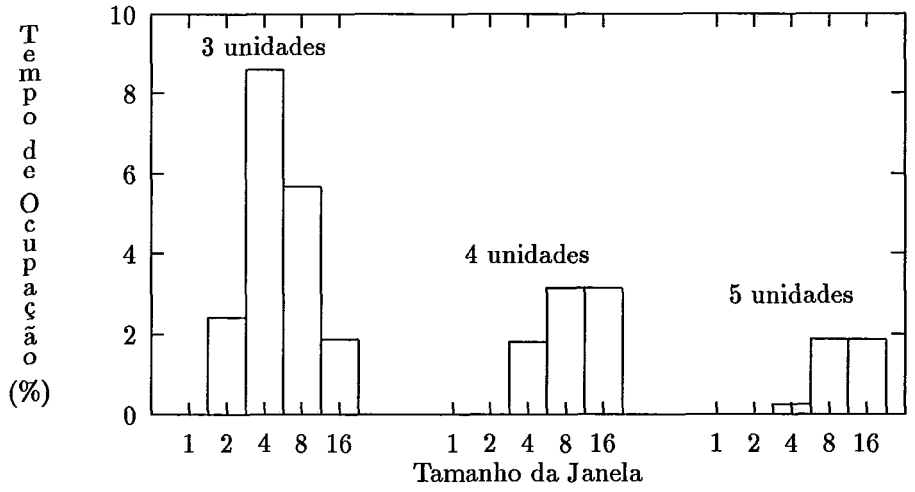

Figura 5.4: Ocupação de Grupos de 3 a 5 Unidades *Cores* no programa *BCDBin*

Quando o tamanho da janela passa para 2 (vide Tabela 5.3), o tempo ocioso que era de 24,95 % sobe para 26,42 %, a porcentagem do tempo de processamento em que somente 1 unidade esteve ocupada cai para 56,90 %, e o total de *Cores* utilizadas cresce para 3 (durante 14,26 % do tempo de processamento, 2 unidades funcionais permaneceram ocupadadas, e apenas durante 2,42 % é que tivemos 3 unidades *Cores* operando em paralelo).

Um novo aumento no tempo ocioso (27,79 %) ocorre quando o tamanho da janela chega a 4 (vide Tabela 5.4). Além disso, podemos observar uma queda na taxa de utilização das unidades *Cores* para janelas com 4 instruções: caem as porcentagens de tempo em que grupos de 1 e 2 unidades permaneceram ocupadas (50,40 % e 11,15 % respectivamente) e sobe para 5 o número de *Cores* utilizadas (durante 8,59 %, 1,82 % e 0,25 % do tempo de execução, grupos de 3, 4, e 5 unidades funcionais ficaram ocupadas).

Finalmente, para janelas com tamanho 8 a percentagem de ociosidade chega a 30,74 %, e as taxas de ocupação de grupos de 1 e 3 unidades decrescem (47,10 % e 5,68 %). Para grupos de 2 unidades, a taxa de ocupação praticamente é mantida (11,20 %), aumentando para grupos de 4 e 5 unidades (3,15 % e 1,87 % respectivamente). Com esse tamanho de janela, o total de unidades utilizadas chega a 6, e durante 0,26 % todas elas estiveram ocupadas.

Estes dados, ilustrados nos histogramas das Figuras 5.3 e 5.4, mostram que no programa *BCDBin* o aumento do tamanho da janela resultou numa pior utilização das unidades *Core*, apesar da taxa de aceleração ter aumentado. Esta conclusão está baseada nos seguintes fatores:

- tendência de crescimento no tempo ocioso;
- tendência de crescimento no tempo de ocupação de grupos de 3, 4 e 5 unidades;
- tendência de queda no tempo de ocupação de grupos de 1 e 2 unidades

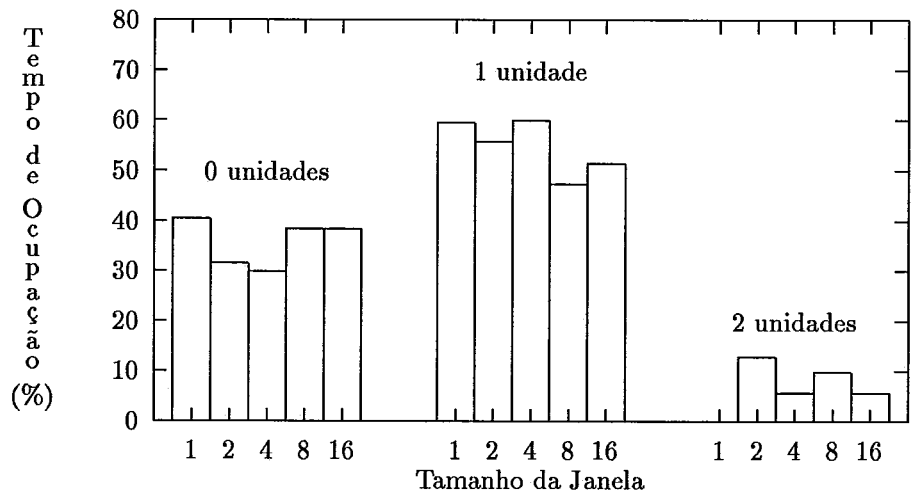

Figura 5.5: Ocupação de Grupos de 0 a 2 Unidades *Cores* no programa *Bubble*

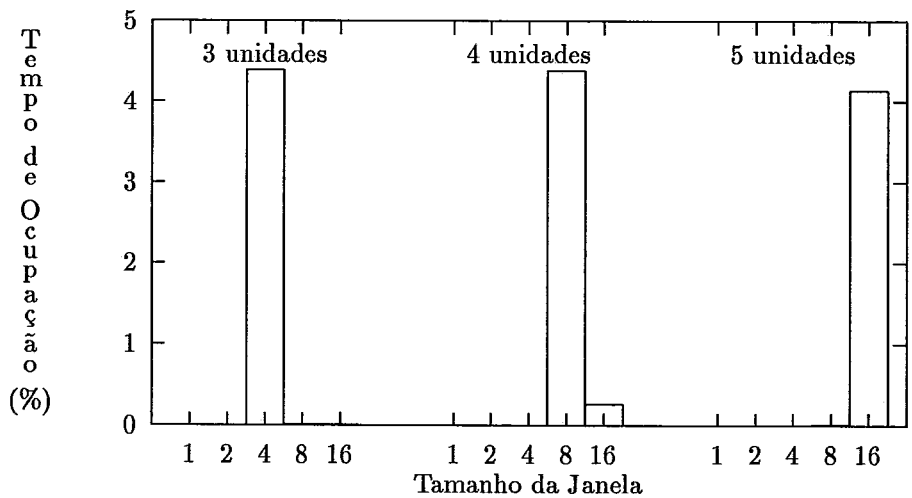

Figura 5.6: Ocupação de Grupos de 3 a 5 Unidades *Cores* no programa *Bubble*

Por outro lado, conforme ilustrado nas Figuras 5.5 e 5.6, as medidas de desempenho produzidas pelo programa *Bubble-Sort* revelam um comportamento diferente. Para uma janela com tamanho 1, a taxa de ociosidade foi de 40,48 %, e somente 1 unidade *Core* foi utilizada (permaneceu ocupada durante 59,92 % do tempo de execução).

Quando o tamanho da janela passa para 2, a porcentagem de ociosidade cai para 31,56 % e 2 unidades funcionais foram utilizadas. Durante 55,65 % do tempo de processamento, somente 1 unidade *Core* permaneceu ocupada, e durante 12,79 % desse tempo as 2 unidades estiveram ocupadas.

Para uma janela com 4 instruções, o nível de ociosidade cai novamente para 29,92 %, a porcentagem de tempo em que somente 1 unidade funcional *Core* esteve ocupada retorna para um patamar bem próximo daquele obtido com uma janela de tamanho 1 (59,93 %) e o número de unidades *Core* utilizadas sobe para 3. Podemos verificar também que durante 5,67 % e 4,39 % do tempo de processamento, grupos de 2 e 3 unidades *Core* permaneceram ocupadas.

No programa *Bubble-Sort*, o aumento na taxa de aceleração decorre de uma melhor utilização dos recursos. Baseamos esta conclusão nos seguintes fatores:

- tendência de queda no tempo ocioso;
- pouca variação na ocupação de grupos de 1 unidade *Core*;
- necessidade de utilização de poucas unidades funcionais (no máximo 3 *Cores* foram ocupadas simultaneamente)

É importante lembrar que tais fatores estão relacionados somente com os resultados obtidos a partir de tamanhos de janela que proporcionaram a elevação das taxas de aceleração: apesar de termos observado a operação simultânea de grupos com 4 e 5 unidades *Core* para janelas com mais do que 4 instruções, as taxas de aceleração correspondentes não aumentaram no caso do programa *Bubble-Sort.*

Através dessas conclusões, podemos fazer algumas considerações acerca da capacidade de armazenamento da janela de instruções de um processador Super Escalar.

No programa *BCDBin* por exemplo, como existe uma tendência de aumento no tempo de ociosidade dos recursos à medida que o tamanho da janela cresce, torna-se necessário avaliar até que ponto é vantajoso aumentar o tamanho da janela e o número de unidades funcionais para melhorar a taxa de aceleração.

Omitindo-se tal avaliação, corremos o risco de elevar os custos de implementação, incorporando recursos que serão subutilizados. Além disso, a necessidade de empregar um grande número de unidades funcionais é questionável. Por exemplo, durante nossos experimentos com esse programa de teste (*BCDBin*), para uma janela com 8 instruções, verificamos que até 6 unidades *Core* foram utilizadas. Contudo, 3 unidades foram suficientes durante 94,72 % do tempo de processamento. Para uma janela com 4 instruções, embora até 5 *Cores* tenham sido utilizadas simultaneamente, 3 unidades foram suficientes durante 97,93 % do tempo de processamento.

Ao reduzir o total de unidades funcionais do processador, poderíamos obter melhores taxas de ocupação. Contudo, essa redução poderia provocar uma significativa queda na eficiência do processador, violando desse modo, um dos principais objetivos de uma arquitetura Super Escalar (i.e., alto desempenho). Por esse motivo, o projetista geralmente ignora o fator custo em favor do desempenho. Alternativamente, ele poderia optar por uma janela com menor capacidade de armazenamento.

Concluímos, portanto, que para determinar o tamanho de janela ideal, precisamos estudar o comportamento das taxas de aceleração e da ociosidade dos recursos que constituem o processador Super Escalar. Apesar de não termos conduzido experimentos com modelos constituídos por um reduzido número de dispositivos funcionais, os estudos aqui apresentados indicaram que janelas com 2 e 4 instruções são as que apresentam o melhor desempenho segundo o critério *custo x benefício* para o *benchmark* utilizado.

Excetuando o programa *Integral*, nossa bateria de teste não apresentou aumentos significativos na taxa de aceleração para janelas com mais do que 4 instruções. Adicionalmente, todos os programas que foram processados na configuração cuja janela armazena 2 instruções, apresentaram taxas de aceleração signi-

ficativamente melhores do que as obtidas no modelo cuja janela armazena 1 única instrução. O tempo ocioso das unidades *Core* reforça esta afirmação pois, excetuando os programas *Branch* e *BCDBin*, todos os programas atingiram a menor taxa de ociosidade para janelas com 2 e 4 instruções.

5.2 Impacto da Utilização de Múltiplos *CDBs*

Na seção anterior apresentamos os estudos que avaliaram o efeito do tamanho da janela de instruções no desempenho da máquina. Para que esse efeito ficasse exclusivamente limitado pelas características dos programas do *benchmark* (dependências verdadeiras de dados e desvios condicionais) foram incorporados, no modelo de máquina, 64 unidade funcionais de cada tipo (cada unidade com sua *reservation station*) e 64 *CDBs*. Desta forma, descartamos a hipótese do mecanismo de despacho ser interrompido pela falta de *reservation stations* livres e evitamos que a propagação de resultados, devido à contenção no uso dos *CDBs*, fosse retardada. Contudo, a baixa utilização dos *CDBs* diante do seu elevado custo e complexidade de implementação (múltiplos barramentos interconectando todos os dispositivos funcionais do processador), motivaram o estudo relacionado com o impacto da variação do número de *CDBs* no desempenho da máquina.

As Tabelas 5.7, 5.8, 5.9, 5.10 e 5.11 apresentam, para cada programa, a percentagem do tempo de processamento em que grupos de *CDBs* permaneceram ocupados em função do tamanho da janela de instruções de uma máquina configurada com 64 *CDBs*. O número máximo de *CDBs* (Total CDBs) que foram simultaneamente utilizados estão listados nessas tabelas.

Examinando os dados apresentados nestas tabelas, verificamos que o total de *CDBs* utilizados cresce com o tamanho da janela de instruções. Este resultado já era esperado, pois o número de instruções tentando propagar seus resultados simultaneamente também tende a crescer com o tamanho da janela, o que aumenta a demanda por *CDBs*. Dessa forma, enquanto que para janelas de tamanho 1, no máximo 2 *CDBs* foram utilizados ao mesmo tempo, para janelas de tamanho 16 este total chega a 8 *CDBs*.

Programa	0 CDBs	1 CDB	2 CDBs	> 2 CDBs	Max CDBs
Integral	78,66	17,31	4,03	———	2
LU	73,33	25,74	0,93	———	2
Hu-Tucker	76,29	23,70	0,01	———	2
Branch	83,62	16,38	———	———	1
BCDBin	76,74	23,25	0,01	———	2
Livermore	74,40	25,60	———	———	1
Quick-Sort	77,29	22,71	———	———	1
Bubble-Sort	77,09	22,91	———	———	1

Tabela 5.7: Ocupação dos *CDBs* para Janelas de Tamanho 1

Programa	0 CDBs	1 CDB	2 CDBs	> 2 CDBs	Max CDBs
Integral	68,37	23,26	7,43	0,94	3
LU	72,20	14,18	11,97	1,65	3
Hu-Tucker	76,24	13,13	10,31	0,32	4
Branch	87,76	4,09	8,15	———	2
BCDBin	78,23	16,21	4,16	1,40	3
Livermore	74,46	19,00	6,54	———	2
Quick-Sort	78,03	14,93	6,77	0,27	3
Bubble-Sort	75,12	18,48	6,39	———	2

Tabela 5.8: Ocupação dos *CDBs* para Janelas de Tamanho 2

Por outro lado, analisando a porcentagem do tempo de ocupação de grupos de *CDBs*, verificamos que a demanda por um número maior destes recursos ocorre durante uma pequena fração do tempo de execução. O programa *Hu-Tucker*, por exemplo, embora chegue a utilizar 8 *CDBs* quando executado na configuração que inclui uma janela com 16 instruções, ocupou somente 2 *CDBs* durante 97,22 % do tempo de execução. Além disso, para qualquer tamanho de janela, 2 *CDBs* são suficientes durante pelo menos 91,74 % do tempo de execução de todos os programas de teste. Esta situação limite ocorre com o programa *Integral* quando executado com uma janela de 8 instruções. Para janelas com 2 e 4 instruções (tamanhos apontados como ideais na seção anterior) durante, pelo menos, 98 % e 94 % do tempo de execução de todos os programas de teste, respectivamente, no máximo 2 *CDBs* foram utilizados simultaneamente. Já nas configurações com janelas de tamanho 1, embora para alguns programas elas tenham requerido 2 *CDBs*, um único barramento foi suficiente durante a maior parte do tempo de processamento: excetuando o programa *Integral*, os componentes do *benchmark* não precisaram de

mais do que 1 *CDB* durante, no mínimo, 99 % do tempo de execução.

Programa	0 CDBs	1 CDB	2 CDBs	> 2 CDBs	Max CDBs
Integral	59,72	28,19	6,73	5,36	4
LU	72,82	14,83	6,87	5,48	5
Hu-Tucker	76,89	12,79	7,13	3,19	5
Branch	87,76	4,09	8,15	———	2
BCDBin	80,05	12,00	4,40	3,55	5
Livermore	74,67	20,26	3,37	1,70	3
Quick-Sort	78,53	11,77	8,67	1,03	4
Bubble-Sort	75,55	17,79	5,20	1,46	3

Tabela 5.9: Ocupação dos *CDBs* para Janelas de Tamanho 4

Programa	0 CDBs	1 CDB	2 CDBs	> 2 CDBs	Max CDBs
Integral	61,17	23,83	6,74	8,26	6
LU	72,87	13,59	10,07	3,47	5
Hu-Tucker	77,53	14,19	4,78	3,50	6
Branch	87,76	4,09	8,15	———	2
BCDBin	79,49	13,13	3,73	3,65	6
Livermore	74,67	21,95	0,01	3,37	4
Quick-Sort	78,12	13,29	5,62	2,97	4
Bubble-Sort	78,39	13,58	6,57	1,46	5

Tabela 5.10: Ocupação dos *CDBs* para Janelas de Tamanho 8

Estes dados indicam que é desnecessário empregar uma quantidade tão grande de *CDBs* como ocorreu com os experimentos apresentados na seção anterior (64 *CDBs* foram empregados). Além disso, configurações com mais de 2 *CDBs* podem provocar a subutilização desses recursos. Contudo, uma vez que em algumas situações até 8 *CDBs* foram ocupados simultaneamente, empregar um número menor de barramentos poderá gerar muitos conflitos no uso do *CDB* (quanto menor o total de *CDBs* empregados, maior a possibilidade de ocorrência de conflito no uso desse recurso).

Dizemos que ocorre conflito quando existem mais instruções tentando utilizar simultaneamente os *CDBs*, do que barramentos disponíveis. Neste caso, um mecanismo que atua como árbitro é ativado, determinando quais instruções têm prioridade no uso dos *CDBs*. As instruções que não foram autorizadas, têm o término de sua execução adiado para o próximo ciclo de máquina, quando serão submetidas ao processo de arbitramento, e se for o caso, a novos adiamentos. Além disso, a

Programa	0 CDBs	1 CDB	2 CDBs	> 2 CDBs	Max CDBs
Integral	56,28	31,93	5,08	6,71	7
LU	72,87	13,59	8,15	5,39	5
Hu-Tucker	78,03	14,54	4,65	2,78	8
Branch	87,76	4,09	8,15	———	2
BCDBin	79,57	13,04	3,73	3,66	7
Livermore	74,67	21,95	0,01	3,37	4
Quick-Sort	78,46	13,20	6,30	2,04	5
Bubble-Sort	78,37	14,96	5,20	1,47	5

Tabela 5.11: Ocupação dos *CDBs* para Janelas de Tamanho 16

execução de uma instrução poderá ser retardada se ela apresentar dependências verdadeiras de dados com outras cujos términos tenham sido adiados. O efeito cumulativo destes múltiplos e sucessivos atrasos pode comprometer o desempenho da máquina, reduzindo consideravelmente as taxas de aceleração. Portanto, a tarefa de avaliação do número ideal de *CDBs* que devem ser empregados em uma máquina, consiste em achar um ponto de equilíbrio entre a tendência de reduzir o número de barramentos (para melhor utilizá-los) e o de conflitos. O estudo do impacto da variação do número de *CDBs* sobre o total de conflitos e da influência destes nas taxas de aceleração, permite determinar tal ponto de equilíbrio.

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	2,86	9,21	27,29	31,76	33,52
LU	0,84	7,54	11,43	13,29	11,72
Hu-Tucker	0,01	5,34	11,78	14,53	15,22
Branch	0,00	7,54	7,54	7,54	7,54
BCDBin	0,01	6,83	10,56	12,48	12,48
Livermore	0,00	1,66	4,98	3,38	3,38
Quick-Sort	0,00	4,79	7,74	9,33	10,47
Bubble-Sort	0,00	4,77	6,51	9,37	9,37

Tabela 5.12: Ciclos (%) em que Houve Conflitos para Modelos com 1 *CDB*

As Tabelas 5.12, 5.13 e 5.14 apresentam, para modelos com 1, 2 e 3 *CDBs*, a porcentagem do tempo de execução em que ocorreram conflitos no uso de *CDBs*.

Estes dados demonstram que ao empregarmos 2 *CDBs* (em vez de 1 único *CDB*), obtivemos reduções significativas na ocorrência de conflitos. Por exemplo, no programa *Hu-Tucker*, quando executado com uma janela com 4 instruções,

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	0,00	0,93	5,37	10,20	12,67
LU	0,01	1,43	2,83	3,02	3,13
Hu-Tucker	0,00	0,32	2,57	3,35	4,49
Branch	0,00	0,00	0,00	0,00	0,00
BCDBin	0,00	1,40	3,51	4,24	4,25
Livermore	0,00	0,00	1,69	1,69	1,69
Quick-Sort	0,00	0,27	1,02	2,24	2,24
Bubble-Sort	0,00	0,00	1,46	1,47	2,84

Tabela 5.13: Ciclos (%) em que Houve Conflitos para Modelos com 2 *CDB*

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	0,00	0,00	1,34	3,00	5,03
LU	0,00	0,00	1,22	1,35	0,46
Hu-Tucker	0,00	0,01	0,44	1,58	2,24
Branch	0,00	0,00	0,00	0,00	0,00
BCDBin	0,00	0,00	1,61	1,74	1,75
Livermore	0,00	0,00	0,00	0,01	0,01
Quick-Sort	0,00	0,00	1,01	0,45	1,05
Bubble-Sort	0,00	0,00	0,00	1,46	1,46

Tabela 5.14: Ciclos (%) em que Houve Conflitos para Modelos com 3 *CDB*

o aumento no número de barramentos proporcionou uma redução de aproximadamente 80 % na ocorrência de conflitos. No programa *Quick-Sort*, com o mesmo tamanho de janela, esta redução chega a quase 90 %. Além disso, ao empregarmos 2 *CDBs* conseguimos reduzir o total de ciclos em que ocorreram conflitos para menos de 5 % do tempo de execução, para todos os programas, a exceção do *Integral*. Esta taxa de conflitos é significativamente baixa, e se levarmos em conta as reduções bem menores proporcionadas por um novo aumento no total de barramentos (3 *CDBs*), podemos concluir que 2 *CDBs* devem ser suficientes. A análise da variação das taxas de aceleração (Tabelas 5.15, 5.16, 5.17 e 5.1) em relação ao número de *CDBs* reforça esta conclusão.

Podemos constatar que, no modelo com 2 *CDBs*, a taxa de aceleração obteve um crescimento, em relação ao modelo com um único *CDB*, de até 18 % para o programa *Integral* e de no máximo 8 % para os demais programas. Nas configurações com 3 *CDBs*, o crescimento da taxa de aceleração (em relação à configuração com 2 *CDBs*) é de até 5 % para o programa *Integral* e não mais do que 2 % para os outros

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	2,10	3,37	4,54	4,85	4,91
LU	1,95	2,99	3,13	3,10	3,14
Hu-Tucker	1,81	2,57	2,73	2,69	2,67
Branch	1,49	1,72	1,72	1,72	1,72
BCDBin	1,72	2,09	2,24	2,29	2,29
Livermore	1,83	2,25	2,25	2,29	2,29
Quick-Sort	1,72	2,18	2,40	2,46	2,45
Bubble-Sort	1,84	2,47	2,61	2,62	2,62

Tabela 5.15: Taxas de Aceleração Obtidas com 1 *CDB*

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	2,11	3,37	4,91	5,34	5,80
LU	1,95	3,04	3,25	3,23	3,27
Hu-Tucker	1,81	2,65	2,81	2,81	2,82
Branch	1,49	1,86	1,86	1,86	1,86
BCDBin	1,72	2,13	2,37	2,45	2,45
Livermore	1,83	2,29	2,29	2,29	2,29
Quick-Sort	1,72	2,22	2,51	2,56	2,59
Bubble-Sort	1,84	2,52	2,62	2,62	2,62

Tabela 5.16: Taxas de Aceleração Obtidas com 2 *CDBs*

programas. Verificamos, portanto, que os ganhos obtidos na taxa de aceleração quando empregamos mais do que 2 barramentos, não são muito significativos. A Figura 5.7 reforça essa afirmação. Nela são apresentadas as taxas de aceleração do programa *BCDBin* para configurações com 1, 2, 3 e 64 *CDBs*, em função do tamanho da janela de instruções. A grande proximidade das taxas relativas aos processadores com 2, 3 e 64 *CDBs*, demonstra ser desnecessário utilizar mais de 2 barramentos.

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	2,11	3,40	4,91	5,50	6,14
LU	1,95	3,05	3,26	3,27	3,31
Hu-Tucker	1,81	2,65	2,83	2,83	2,85
Branch	1,49	1,86	1,86	1,86	1,86
BCDBin	1,72	2,13	2,44	2,50	2,50
Livermore	1,83	2,29	2,29	2,29	2,29
Quick-Sort	1,72	2,22	2,51	2,58	2,60
Bubble-Sort	1,84	2,52	2,62	2,62	2,62

Tabela 5.17: Taxas de Aceleração Obtidas com 3 *CDBs*

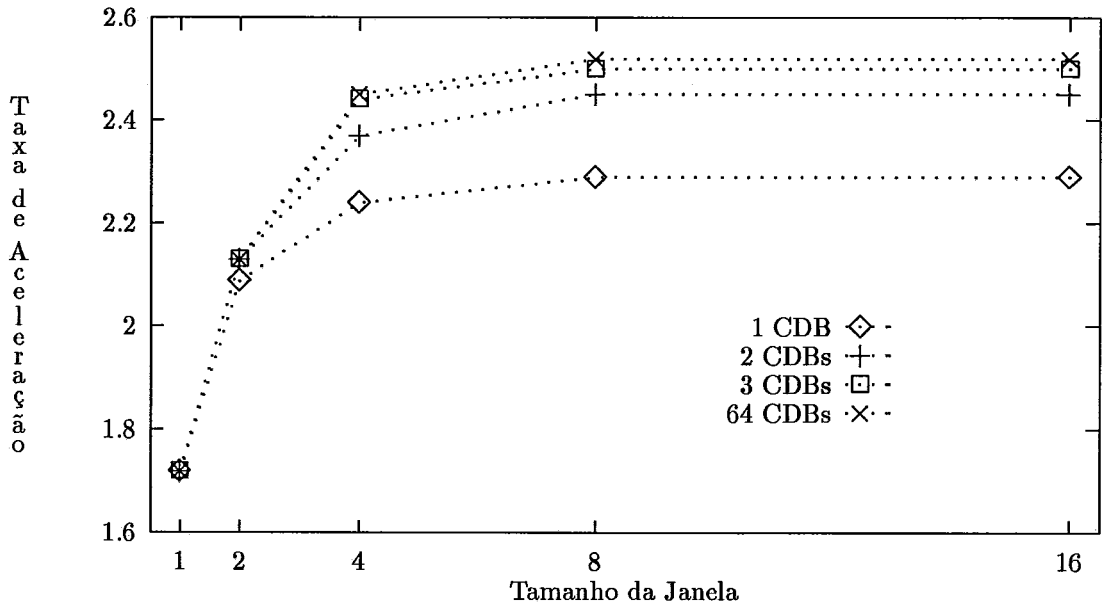

Figura 5.7: Efeitos dos *CDBs* e das Janelas de Instruções no Desempenho

Por outro lado, como ilustrado também na Figura 5.7, somente para janelas com tamanho maior ou igual a 4 instruções é que conseguimos identificar grandes diferenças entre as taxas de aceleração obtidas com 1 e 2 *CDBs*. Para janelas com 2 instruções, à exceção do programa *Branch*, nenhum outro apresentou ganhos maiores que 3 % na taxa de aceleração, quando foram empregados 2 *CDBs* em vez de 1 barramento. Para janelas com tamanho 4, a exceção do *Integral* e do *Branch* (eles apresentaram cerca de 8 %), nenhum outro programa conseguiu ganhos maiores do que 6 %. Finalmente, nas janelas com 16 instruções, como mencionado anteriormente, obtivemos ganhos de 18 % para o programa *Integral* e no máximo de 8 % para os demais. Concluímos, portanto, que a incidência de conflitos nos experimentos com somente 1 *CDB* não gera efeitos significativamente negativos sobre as taxas de aceleração. Assim, embora múltiplos *CDBs* possam eventualmente eliminar os conflitos, o ganho no desempenho não justifica a complexidade e os custos da implementação de vários barramentos na máquina. O insignificante impacto da variação do número de *CDBs* sobre as taxas de aceleração está ilustrado na Figura 5.8, que mostra o desempenho dos programas executados em configurações cujas janelas armazenam 4 instruções ante a variação do número de *CDBs*.

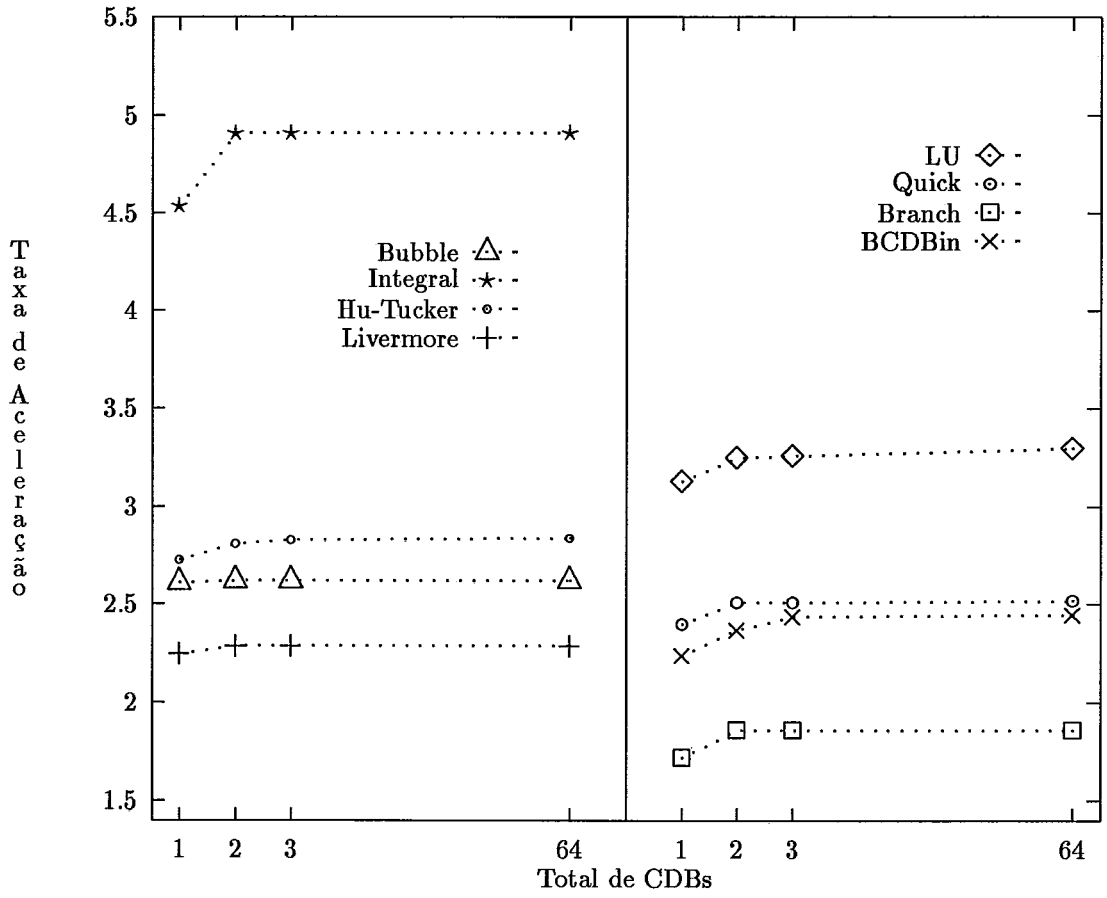

Figura 5.8: Efeito dos *CDBs* nas Taxas de Aceleração

5.3 Influência das *Reservation Stations*

O conceito de *reservation station* foi proposto por Tomasulo em [TOMA67]. Como descrito no Capítulo 2, as *reservation stations* realizam a função de armazenamento das instruções despachadas. Quando as *reservation stations* são incorporadas à arquitetura de uma máquina, as unidades funcionais somente executam as instruções, ficando as *reservation stations* encarregadas pelo armazenamento dessas instruções após o despacho. Ao utilizarmos *reservation stations*, podemos reduzir o tempo de bloqueio da tarefa de despacho pela ausência de recursos disponíveis para receber as instruções, sem contribuir para a subutilização das unidades funcionais.

Os experimentos realizados com os modelos de máquina prévios, demonstraram a importância da utilização de *reservation stations* no desempenho de

uma arquitetura Super Escalar [FER92b]. Nos experimentos realizados com o presente modelo de máquina, como estávamos interessados em avaliar a eficácia do algoritmo de despacho num processador com recursos ilimitados, não variamos o número de *reservation stations* nem o de unidades funcionais, o que teria sido ideal para avaliar a importância das *reservation stations* no desempenho da máquina. Contudo, a comparação do tempo de ocupação de grupos de *reservation stations* e de unidades funcionais deste modelo de máquina, permitem avaliar a influência dessas unidades virtuais. Nesta seção, utilizando essas comparações, estudaremos o efeito das *reservation stations* no desempenho do nosso modelo de máquina Super Escalar.

As Tabelas 5.18, 5.19, 5.20, 5.21 e 5.22 apresentam a porcentagem do tempo de execução dos programas de teste em que grupos de *reservation stations* associadas às unidades funcionais *Core* permaneceram ocupadas para configurações cujas janelas armazenam 1, 2, 4, 8 e 16 instruções, respectivamente.

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	52,14	47,86	—	—	—	—	—
LU	36,18	63,67	0,10	0,03	0,01	0,01	—
Hu-Tucker	38,69	61,23	0,05	0,03	—	—	—
Branch	24,63	75,37	—	—	—	—	—
BCDBin	24,85	75,05	0,01	—	—	—	—
Livermore	32,95	64,03	2,01	1,01	—	—	—
Quick-Sort	29,79	69,71	0,50	—	—	—	—
Bubble-Sort	40,34	59,52	0,14	—	—	—	—

Tabela 5.18: Ocupação (%) das *RSs* da *Core* para Janelas de Tamanho 1

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	28,35	46,56	22,30	2,79	—	—	—
LU	11,96	18,56	29,32	21,39	15,78	1,15	1,84
Hu-Tucker	14,15	39,07	38,44	7,50	0,74	0,02	0,07
Branch	30,66	32,63	36,70	0,01	—	—	—
BCDBin	16,60	30,25	50,51	2,64	—	—	—
Livermore	10,24	23,26	43,09	20,25	3,16	—	—
Quick-Sort	18,07	29,32	44,86	6,95	0,80	—	—
Bubble-Sort	13,97	37,07	39,51	9,45	—	—	—

Tabela 5.19: Ocupação (%) das *RSs* da *Core* para Janelas de Tamanho 2

Para configurações cujas janelas armazenam uma instrução, a com-

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	12,71	46,99	8,14	12,07	12,05	8,04	———
LU	8,04	12,40	7,98	25,41	2,04	7,54	36,59
Hu-Tucker	14,37	16,81	27,19	26,05	8,43	0,58	6,57
Branch	30,65	32,64	12,23	12,24	12,23	0,01	———
BCDBin	19,16	20,74	27,63	21,79	5,55	5,13	———
Livermore	10,25	10,61	16,47	24,06	28,48	10,11	0,02
Quick-Sort	18,62	22,06	29,05	9,91	16,71	1,82	1,83
Bubble-Sort	10,40	15,85	24,36	25,83	23,56	——	———

Tabela 5.20: Ocupação (%) das *RSs* da *Core* para Janelas de Tamanho 4

paração entre as tabelas 5.18 e 5.2 mostra diferenças insignificantes entre as taxas de ocupação de grupos de *Cores* e de grupos de *reservation stations* associadas à este tipo de unidade funcional. A exceção dos programas *Livermore*, *LU* e *Quick-Sort*, onde esta diferença chega a 2,05 %, 1,06 % e 0,5 % respectivamente, em nenhum programa foi registrada uma diferença maior do que 0,14 % entre as taxas de ocupação. Verificamos, também, que é desnecessário associar mais do que 1 *reservation station* às unidades *Core*, pois a exceção do *Livermore*, nenhum programa utilizou 2 ou mais destes dispositivos simultaneamente durante mais de 0,5 % do tempo de execução. Assim, verificamos que para janelas com 1 instrução, é desnecessário associar mais de 1 *reservation station* às *Cores*.

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	14,60	43,47	9,36	9,35	4,65	4,63	13,94
LU	7,90	12,38	7,00	19,28	2,81	5,49	45,14
Hu-Tucker	14,24	16,90	26,63	13,43	8,95	5,42	14,43
Branch	30,65	32,64	12,23	12,24	12,23	0,01	———
BCDBin	19,71	18,46	30,58	11,98	1,88	12,11	2,88
Livermore	10,24	10,60	15,21	15,21	23,43	10,13	15,18
Quick-Sort	19,10	22,12	27,21	5,31	8,41	5,37	12,48
Bubble-Sort	10,38	11,72	19,97	11,21	31,13	15,59	———

Tabela 5.21: Ocupação (%) das *RSs* da *Core* para Janelas de Tamanho 8

Para janelas com 2 ou mais instruções, as diferenças entre as taxas de ocupação de grupos de *reservation stations* e unidades funcionais, e entre os totais utilizados destes recursos, tendem a ser maiores e mais significativas. O programa *LU*, por exemplo, quando executado numa configuração cuja janela contém 2 instruções, precisou no máximo de 3 *Cores* (Tabela 5.3), enquanto que este mesmo

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	15,86	48,76	10,14	5,11	0,02	5,02	15,09
LU	7,91	12,38	7,00	19,28	2,65	5,41	45,37
Hu-Tucker	14,24	16,71	26,88	14,76	9,06	1,05	17,30
Branch	30,65	32,64	12,23	12,24	12,23	0,01	———
BCDBin	19,71	18,46	30,84	11,72	1,88	12,11	5,28
Livermore	10,25	10,60	15,21	15,21	23,43	10,13	15,17
Quick-Sort	19,24	20,06	28,91	7,56	6,26	7,11	10,86
Bubble-Sort	10,38	11,72	19,97	11,21	27,01	9,60	10,11

Tabela 5.22: Ocupação (%) das *RSs* da *Core* para Janelas de Tamanho 16

número de *reservation stations* foi suficiente durante 81,23 % do tempo de execução (Tabela 5.19). O programa *Livermore*, para este mesmo tamanho de janela, utilizou somente 2 unidades *Core* durante a execução, porém precisou de 3 ou mais *reservation stations* deste tipo durante 23,41 % do tempo. Quando executado com uma janela de tamanho 4, o programa *Integral* precisou de 4 ou mais *reservation stations* durante 20,05 % do tempo (Tabela 5.20), enquanto que 3 *Cores* foram suficientes durante toda a execução (Tabela 5.4). Já o programa *LU* (executado na configuração cuja janela armazena 4 instruções) ocupou no máximo 4 *Cores* e precisou de 5 ou mais *reservation stations* durante 44,13 % do tempo de execução.

Porém, ao considerarmos a possibilidade de empregar um número de unidades *Core* menor do que o máximo utilizado (para obter uma melhor ocupação destes recursos), é que a importância das *reservation stations* se torna mais clara. Por exemplo, nos programas *Integral*, *Livermore* e *Bubble-Sort*, para janelas com 2 instruções, 1 *Core* foi suficiente durante 86,05 %, 89,86 % e 87,21 % do tempo de execução respectivamente, enquanto que um número equivalente de *reservation stations* seria suficiente respectivamente durante somente 74,91 %, 33,50 % e 51,04 % do tempo de processamento. Comparação idêntica para janelas com 4 instruções indica que nos programas *Integral*, *LU*, *Hu-Tucker*, *BCDBin* e *Quick-Sort*, 2 unidades *Core* foram suficientes durante 87,93 %, 88,23 %, 94,67 %, 89,34 % e 96,94 % do tempo de execução, enquanto que um número igual de *reservation stations* foi suficiente durante somente 67,84 %, 28,42 %, 58,37 %, 67,53 % e 69,73 %, respectivamente.

Mesmo assim, a importância das *reservation stations* ante a utilização de um número reduzido de unidades funcionais, se apresenta subestimada nestes

dados. Caso simulássemos configurações com um número de unidades *Core* menor do que o máximo utilizado, o potencial de exploração de paralelismo entre instruções seria menor e estas teriam que aguardar mais tempo nas *reservation stations*, o que tenderia a aumentar a taxa de ocupação e o total utilizado destes dispositivos. Uma vez que nosso modelo só possui 1 unidade de Acesso à Memória, a análise da ocupação das *reservation stations* associadas à esse tipo de unidade, serve melhor para avaliar a importância das *reservation stations* em situações em que há um número reduzido de unidades funcionais.

A Tabela 5.23 mostra, para cada programa, a taxa de ocupação da unidade de Acesso à Memória de acordo com o tamanho da janela de instruções.

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	17,36	28,02	40,44	46,73	50,63
LU	25,07	39,13	42,33	42,40	42,40
Hu-Tucker	29,34	42,95	46,03	46,29	46,29
BCDBin	11,66	14,40	16,63	17,11	17,11
Livermore	27,82	34,87	34,87	34,87	34,87
Quick-Sort	17,67	22,78	25,83	26,50	26,69
Bubble-Sort	33,17	45,27	47,16	47,17	47,17

Tabela 5.23: Ocupação (%) da Unidade de Acesso à Memória

As Tabelas 5.24, 5.25, 5.26, 5.27 e 5.28 mostram, para cada programa, as taxas de ocupação de grupos de *reservation stations* associadas à única unidade de Acesso à Memória para janelas armazenando 1, 2, 4, 8 e 16 instruções respectivamente.

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	80,91	17,33	1,76	—	—	—	—
LU	71,61	25,97	2,34	0,08	—	—	—
Hu-Tucker	70,65	29,05	0,24	0,05	0,01	—	—
BCDBin	88,34	11,64	0,02	—	—	—	—
Livermore	71,16	26,78	2,05	0,01	—	—	—
Quick-Sort	82,33	17,55	0,04	0,08	—	—	—
Bubble-Sort	66,82	33,12	0,05	0,01	—	—	—

Tabela 5.24: Ocupação (%) das *RSs* da *Memória* para Janelas de Tamanho 1

Comparando os dados das Tabelas 5.23 e 5.24 podemos ver que as percentagens de tempo em que somente 1 *reservation station* ficou ocupada e as

taxas de ocupação da unidade de acesso à memoria, para janelas de tamanho 1, são bastante semelhantes. A diferença entre esses percentuais é de no máximo 1,04 %. Além disso, a Tabela 5.24 mostra que foi necessário empregar duas ou mais *reservation stations* durante 2,42 % do tempo de execução (no máximo) o que é uma parcela muito pequena. Como também foram feitas observações semelhantes à essas na análise da ocupação das *reservation stations* associadas às unidades *Core*, podemos concluir que, para janelas contendo 1 instrução, é desnecessário incorporar *reservation stations* à máquina. As instruções podem ser despachadas diretamente para as unidades funcionais que, nesse caso, passarão a realizar também as funções de uma *reservation station.*

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	50,60	33,52	15,86	0,02	——	——	——
LU	28,94	44,61	14,02	7,49	4,51	0,37	0,06
Hu-Tucker	36,04	40,31	20,27	3,00	0,14	0,10	0,14
BCDBin	71,56	25,88	2,52	0,04	——	——	——
Livermore	34,75	52,51	10,19	2,55	——	——	——
Quick-Sort	65,40	24,73	9,27	0,60	——	——	——
Bubble-Sort	36,42	24,77	34,82	3,99	——	——	——

Tabela 5.25: Ocupação (%) das *RSs* da *Memória* para Janelas de Tamanho 2

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	0,55	37,74	44,26	10,76	6,69	——	——
LU	13,74	24,06	20,41	11,09	6,80	12,78	11,12
Hu-Tucker	31,67	23,66	13,44	10,21	7,03	6,23	7,76
BCDBin	61,12	30,77	7,97	0,14	——	——	——
Livermore	25,88	34,20	26,58	13,34	——	——	——
Quick-Sort	54,47	23,64	12,93	4,53	4,34	0,09	——
Bubble-Sort	32,15	16,35	19,78	15,23	8,25	8,24	——

Tabela 5.26: Ocupação (%) das *RSs* da *Memória* para Janelas de Tamanho 4

Para configurações cujas janelas armazenam 2 instruções, fica evidente que é necessário empregar um número maior de *reservation stations* do que unidades funcionais e portanto, é demonstrada a importância das *reservation stations.* A parcela do tempo de execução em que duas ou mais *reservation stations* permaneceram ocupadas chega a 38,81 % para o *Bubble-Sort* e, excetuando os programas *BCDBin* e *Quick-Sort*, essa porcentagem nunca é menor que 12,74 %. Este comportamento dos programas *BCDBin* e *Quick-Sort* já era esperado pois ambos

possuem poucas instruções de Acesso à Memória nos seus códigos dinâmicos quando comparadas com as dos demais programas.

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	0,46	23,58	29,52	23,24	7,76	10,81	4,63
LU	13,56	11,91	17,79	10,21	7,65	13,25	25,63
Hu-Tucker	31,66	17,26	12,54	11,61	4,21	4,72	18,00
BCDBin	59,99	29,52	9,81	0,68	——	——	——
Livermore	25,88	25,33	24,07	19,65	5,06	0,01	——
Quick-Sort	47,69	28,04	8,65	7,01	8,46	0,15	——
Bubble-Sort	32,12	10,35	19,80	17,10	8,27	4,12	8,24

Tabela 5.27: Ocupação (%) das *RSs* da *Memória* para Janelas de Tamanho 8

Programa	0 RSs	1 RS	2 RSs	3 RSs	4 RSs	5 RSs	> 5 RSs
Integral	0,50	22,24	26,95	25,17	5,05	5,02	15,07
LU	13,56	11,91	13,91	10,30	4,22	12,96	33,14
Hu-Tucker	31,66	17,26	11,81	9,43	3,40	4,97	21,47
BCDBin	59,99	27,39	11,68	0,94	——	——	——
Livermore	25,88	25,33	19,02	19,65	10,11	0,01	——
Quick-Sort	47,32	28,24	6,50	4,62	12,23	1,09	——
Bubble-Sort	32,12	10,35	19,80	17,10	4,14	4,12	12,36

Tabela 5.28: Ocupação (%) das *RSs* da *Memória* para Janelas de Tamanho 16

Para configurações cujas janelas armazenam 4 instruções, a importância das *reservation stations* é ainda maior. São ocupadas 2 ou mais *reservation stations* durante até 62,20 % do tempo de execução (*LU*). Excetuando os programas *BCDBin* e *Quick-Sort*, nenhum programa precisou utilizar um número maior de *reservation stations* do que unidades funcionais durante menos de 39,92 % do tempo de execução (*Livermore*).

Pelo exposto, concluímos que a importância das *reservation stations* está intimamente relacionada com o número de unidades funcionais da máquina. Quando provemos tantas unidades *Core* quanto necessário, vimos que são poucos os casos em que obtivemos taxas de ocupação significativas para um número de *reservation stations* maior que o total de *Cores* utilizadas. Contudo, tivemos indicações de que se o total de unidades *Core* fosse reduzido este quadro se reverteria.

Esta hipótese é reforçada pela avaliação das taxas de ocupação das *reservation stations* associadas à unidade de Acesso à Memória (vide Tabelas 5.24,

5.25, 5.26, 5.27 e 5.28). Tendo em vista que existe somente uma unidade deste tipo, foi necessário associar 2 ou mais *reservation stations* quando a janela era capaz de armazenar mais do que 1 instrução.

Portanto, quanto maior o tamanho da janela de instruções e menor o número de unidades funcionais, maior a importância das *reservation stations.* Quando for empregada uma janela com somente 1 instrução é desnecessário empregar *reservation stations.*

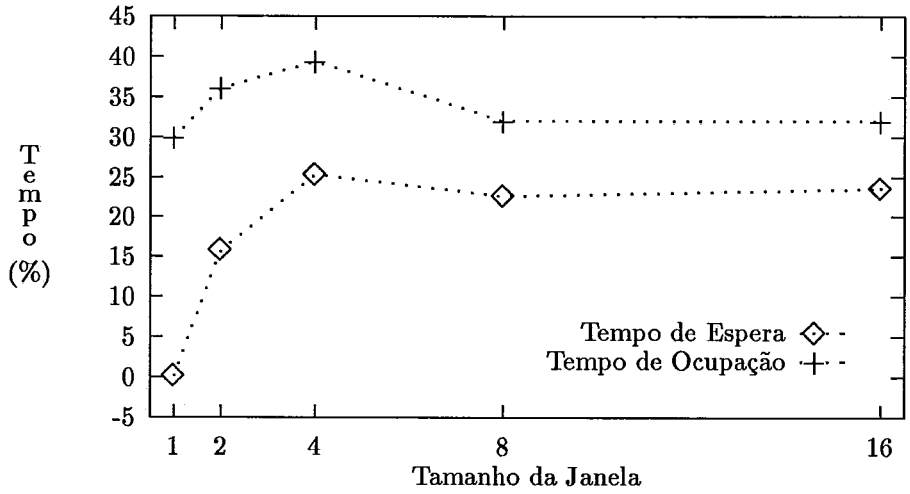

Figura 5.9: Tempo de Espera X Tempo de Ocupação

Embora as Tabelas 5.27, 5.28, 5.21 e 5.22, apresentem taxas de ocupação que, a primeira vista, parecem demonstrar uma maior importância das *reservation stations* quando são empregadas configurações cujas janelas armazenam 8 ou mais instruções, esta impressão é falsa. Como visto na Figura 5.1, a exceção do programa *Integral*, nenhum outro componente do *benchmark* proporcionou ganhos significativos na taxa de aceleração quando empregamos janelas de tamanho maior que 4. Isto se deve, entre outros fatores, às dependências verdadeiras de dados entre instruções. Assim, para janelas com estes tamanhos, as instruções precisam ficar, em média, mais tempo esperando nas *reservation stations* até que os dados, dos quais elas dependem para serem iniciadas, estejam prontos. Daí as altas taxas de ocupação e o grande número de *reservation stations* utilizadas.

A Figura 5.9 mostra o gráfico do tempo médio de espera das *re-*

servation stations e do tempo médio de ocupação das mesmas, quando variamos o tamanho da janela de instruções. Os dados da figura referem-se as *reservation stations* das unidades *Core* durante a execução do programa *Bubble-Sort*. Definimos tempo de espera como sendo o intervalo de tempo em que a *reservation station* ficou ocupada, sem que a instrução nela armazenada fosse iniciada devido a dependências verdadeiras de dados.

5.4 Impacto dos Desvios Condicionais

Na Tabela 5.1 (Seção 5.1) verificamos que é possível obter taxas de aceleração de até 6,14 (programa *Integral*). Infelizmente, este resultado é uma exceção, pois para a maior parte dos programas da bateria de testes obtivemos taxas entre 2,29 e 2,85. Constatamos também (vide Figura 5.1) que as taxas de aceleração tendem a se estabilizar após a janela atingir o tamanho 2 ou 4. Dentre os possíveis fatores que limitam o crescimento da taxa de aceleração podemos destacar:

(i) ausência de recursos disponíveis;

(ii) dependências de dados;

(iii) desvios condicionais

Tendo em vista que foram providas 64 *reservation stations*, 64 unidades funcionais e 64 *CDBs* e que estes recursos nunca foram completamente ocupados durante os experimentos, o fator (i) não afetou o desempenho.

Já o fator (ii) influenciou nossas taxas de aceleração. Ao permitir a execução antecipada (i.e., fora de ordem) de instruções que apresentam dependências falsas, o algoritmo de Tomasulo minimiza o efeito dessa influência. Como é impossível desprezar as dependências verdadeiras de dados, a influência do fator (ii) já está otimizada e não há como reduzí-la.

Finalmente, o fator (iii) limita o desempenho do algoritmo de despacho. Ele tem que ser interrompido após o escalonamento de uma instrução de

desvio condicional, até que seja possível avaliar se a condição (para que o desvio seja tomado) é verdadeira ou não. Uma vez que o algoritmo de Tomasulo não provê mecanismos para reduzir o impacto dos desvios condicionais, decidimos avaliar o efeito dessas instruções.

A Tabela 5.29 mostra, para cada programa, a porcentagem do tempo de execução em que o mecanismo de despacho ficou parado devido ao escalonamento de um desvio condicional. Tais porcentagens representam o somatório das parcelas de tempo entre o despacho de uma instrução de desvio condicional e o término da execução da mesma.

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	1,73	11,15	24,12	35,63	40,26
LU	5,40	24,45	56,13	74,98	83,57
Hu-Tucker	9,19	29,43	57,50	73,16	79,75
Branch	21,29	38,74	63,20	63,21	63,21
BCDBin	14,62	41,79	59,01	75,23	80,24
Livermore	9,92	39,01	69,39	84,58	89,64
Quick-Sort	13,45	40,21	57,76	73,07	76,03
Bubble-Sort	8,36	35,21	61,03	75,29	85,39

Tabela 5.29: Influência dos Desvios Condicionais

Ao examinar as porcentagens da Tabela 5.29, podemos constatar que para janelas de tamanho 2, esta parcela de tempo já é muito elevada, atingindo o patamar de 41,79 % para o programa *BCDBin*. Para janelas com 4 instruções a parcela corresponde a 69,39 % (*Livermore*), no máximo, e para janelas de tamanho 16 chega até a 89,64 % (*Livermore*) do tempo de execução.

A partir desses resultados podemos verificar que é grande o impacto dos desvios condicionais no desempenho de um processador Super Escalar. Isto nos levou a conceber um outro modelo de máquina incorporando um mecanismo de predição de desvios [JLEE84] para reduzir tal impacto. Nos experimentos subseqüentes, adotamos um mecanismo de predição de desvios que pode ser visto como um oráculo oniciente: ele sempre sabe se a transferência de controle irá ocorrer ou não. Desta forma, o mecanismo elimina completamente os retardos impostos no algoritmo de despacho pelas instruções de desvio condicional.

Com o objetivo de eliminar a influência provocada pela falta de recursos utilizamos uma configuração com um número ilimitado de *CDBs*, *reservation stations* e unidades funcionais. Assim, com o desempenho limitado somente pelas dependências verdadeiras de dados, espera-se que um número maior de instruções estejam prontas para serem executadas a cada ciclo, aumentando o nível de ocupação dos dispositivos funcionais e as taxas de aceleração. Sem as limitações impostas pelos desvios condicionais espera-se, também, que seja vantajoso empregar janelas com mais instruções.

A Tabela 5.30 mostra, para cada programa teste, as taxas de aceleração obtidas nesse modelo alternativo.

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	2,34	4,49	8,58	11,93	11,95
LU	2,07	4,06	5,00	5,01	5,01
Hu-Tucker	2,00	3,88	5,11	5,15	5,15
Branch	2,02	3,79	7,56	7,56	7,56
BCDBin	2,02	3,81	6,69	12,42	14,66
Livermore	2,03	3,76	3,81	3,81	3,81
Quick-Sort	2,03	3,87	7,09	9,17	9,17
Bubble-Sort	2,01	3,89	4,61	4,61	4,61

Tabela 5.30: Taxas de Aceleração Obtidas com Predição de Desvios

A Figura 5.10 mostra as taxas de aceleração no modelo com predição de desvios em função do aumento do tamanho da janela de instruções.

Comparando as Figuras 5.10 e 5.1 e as Tabelas 5.30 e 5.1, constatamos o grande impacto dos desvios condicionais sobre as taxas de aceleração. O programa *BCDBin* é o melhor exemplo disto. Anteriormente, sua taxa de aceleração ocupava o sexto lugar dentre os componentes do *benchmark*. Com o mecanismo de predição de desvios ele passou a ser o programa com a maior taxa de aceleração. As taxas de aceleração desse programa chegam a ser quase 6 vezes maiores do que as que foram obtidas no modelo sem predição de desvios. O tempo de processamento do programa *Branch* também apresentou uma redução marcante. Ele que antes era o programa com a pior taxa de aceleração, passou para o quarto lugar. Suas taxas de aceleração são até 4 vezes maiores quando comparadas com as do modelo anterior. Outro programa com notável desempenho é o *Quick-Sort*. Este

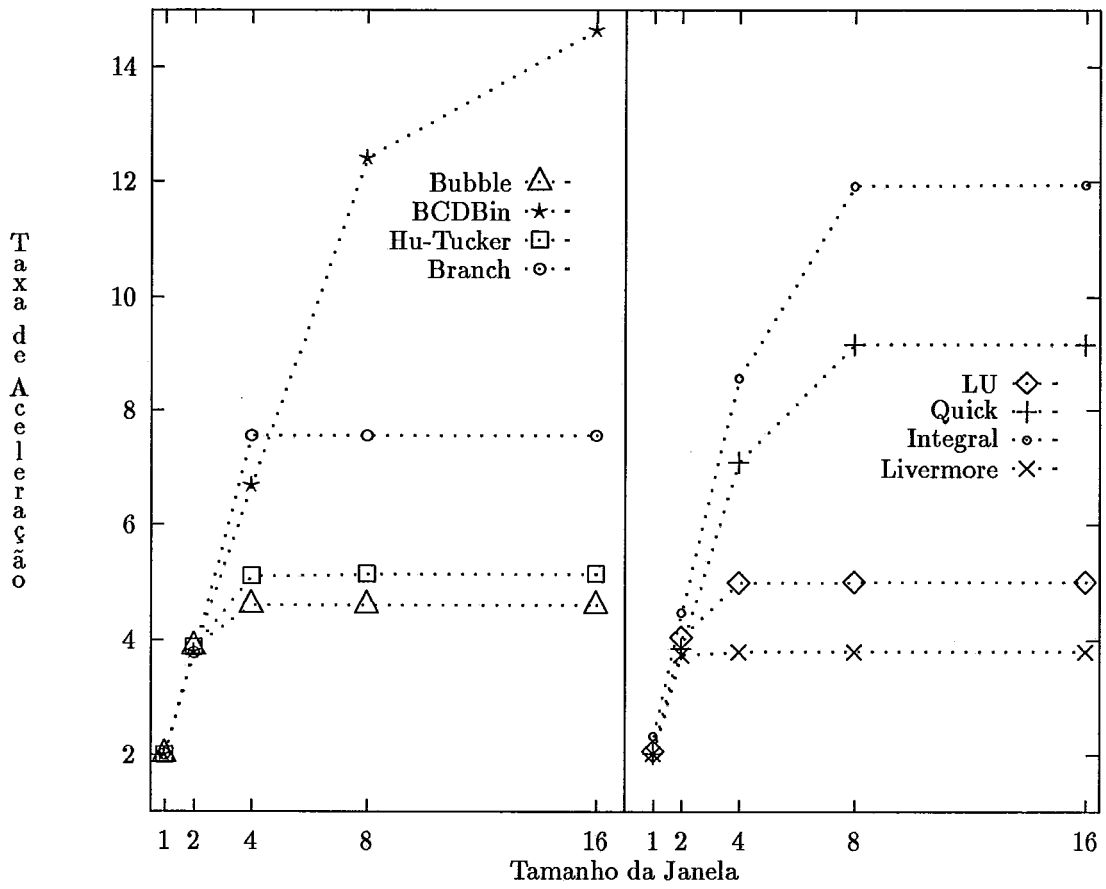

Figura 5.10: Efeito da Predição de Desvios nas Taxas de Aceleração

programa apresentou taxas de aceleração até 3,5 vezes maiores que as obtidas pelo modelo sem mecanismo de predição de desvios. Embora tenha sido o programa que manteve a tarefa de despacho interrompida durante a maior parcela do tempo de execução (89,64 %), o programa *Livermore* foi o que apresentou menores ganhos com a adoção do mecanismo de predição de desvios. A elevada incidência de dependências verdadeiras de dados é a responsável por esse baixo desempenho. Ainda assim, as taxas de aceleração obtidas são até 66% maiores que as anteriores.

Com relação ao tamanho da janela de instruções, verificamos que ao eliminarmos o impacto dos desvios condicionais nas taxas de aceleração, a utilização de janelas de tamanho maior (até 8 instruções) foi vantajosa. A exceção do *Livermore*, todos os programas obtiveram ganhos significativos na taxa de aceleração ao utilizar janelas de tamanho 4. Além disso, compensa utilizar janelas com 8 instruções

na execução dos programas *Integral*, *Quick-Sort* e *BCDBin*. Este comportamento difere do observado no modelo sem mecanismo de predição de desvios (Tabela 5.30 e Figura 5.10). As taxas de aceleração obtidas naqueles experimentos indicaram que era desnecessário utilizar janelas de tamanho maior do que 4. Ainda assim, os ganhos obtidos com janelas com 4 instruções, embora significativos em alguns casos, são bem inferiores que os apresentados na Tabela 5.30.

Programa	1 Instr	2 Instr	4 Instr	8 Instr	16 Instr
Integral	46,94	18,99	1,28	3,42	3,43
LU	34,28	24,55	20,18	21,53	21,32
Hu-Tucker	33,59	14,81	11,89	10,92	11,09
Branch	0,08	0,15	0,27	0,23	0,23
BCDBin	13,54	15,00	0,42	0,72	1,48
Livermore	28,44	18,78	14,67	14,86	14,84
Quick-Sort	20,47	16,01	1,89	15,08	19,53
Bubble-Sort	36,23	20,71	19,33	18,54	17,77

Tabela 5.31: Tempo Ocioso (%) das Unidades *Core*

A Tabela 5.31 mostra a parcela do tempo de execução em que nenhuma unidade *Core* foi ocupada (tempo ocioso) para o modelo que emprega o mecanismo de predição de desvios. Comparando estes dados com aqueles apresentados nas Tabelas 5.2, 5.3, 5.4, 5.5 e 5.6, verificamos a grande contribuição deste tipo de mecanismo para a ocupação dos recursos. Para janelas de tamanho 4, por exemplo, enquanto que o mecanismo de predição de desvios proporciona uma parcela de tempo ocioso de no máximo 20,18 %, quando esse mecanismo não é empregado o tempo ocioso das unidades *Core* é de pelo menos 27,79 %. Além disso, é notável que 4 desses programas mantenham pelo menos 1 de suas unidades *Core* ocupadas no mínimo durante 98,11 % do tempo de execução.

Por outro lado, as taxas de ocupação de grupos de unidades *Core* demonstram haver uma maior demanda por unidades deste tipo quando tal modelo é utilizado. Podemos constatar isso ao comparar os dados das Tabelas 5.32 e 5.4. A Tabela 5.32 mostra a taxa de ocupação de grupos de unidades *Core*, para janelas de 4 instruções, quando o mecanismo de predição de desvios é empregado. Nela verificamos que 2 *Cores* foram suficientes durante 78,87%, 80,91%, 80,10%, 50,19%, 58,7%, 89,89%, 50,94% e 86,92% do tempo de execução, para cada um dos

Programa	0 Cores	1 Core	2 Cores	3 Cores	4 Cores	5 Cores	> 5 Cores
Integral	1,28	30,68	46,91	21,10	0,03	———	———
LU	20,18	31,64	29,09	14,82	3,58	0,39	0,30
Hu-Tucker	11,89	35,99	32,22	13,62	4,02	0,94	1,32
Branch	0,27	0,04	49,88	24,88	24,90	0,03	———
BCDBin	0,42	17,88	40,40	29,27	11,24	0,74	0,05
Livermore	14,67	46,30	28,92	9,73	0,33	0,01	0,04
Quick-Sort	1,89	16,77	32,28	33,06	11,98	0,89	3,13
Bubble-Sort	19,33	36,30	31,29	11,87	0,69	0,25	0,27

Tabela 5.32: Ocupação (%) de Grupos de *Core* com Predição de Desvios

programas relacionados na tabela, respectivamente. Já na Tabela 5.4, que mostra as taxas de ocupação correspondentes (no modelo sem mecanismo de predição de desvios), verificamos que 2 unidades *Core* foram suficientes durante 87,93%, 88,23%, 94,67%, 100%, 89,34%, 99,99%, 96,94% e 95,61% respectivamente. Isto demonstra que o impacto no desempenho da máquina provocado pelo emprego de 3 ou mais unidades *Core*, seria maior se um mecanismo de predição fosse utilizado.

Considerando a possibilidade de empregar um número menor de recursos, tal impacto seria ainda maior. A Tabela 5.32 mostra que 1 *Core* foi suficiente durante somente 31,96%, 51,82%, 47,88%, 0,31%, 18,3%, 60,97%, 18,66% e 55,63% do tempo de execução, para cada um dos programas respectivamente. Para o modelo sem predição de desvios, 1 *Core* foi suficiente durante 71,81%, 75,22%, 75,74%, 75,53%, 78,19%, 84,8%, 73,06% e 89,85% para cada programa respectivamente. Mais do que a importância, essas porcentagens mostram que precisamos de um maior número de unidades *Core* quando o modelo incluir um mecanismo de predição de desvios.

Contudo, os componentes da máquina que apresentaram um maior crescimento de demanda foram as *reservation stations*. Como mencionado anteriormente, empregamos tantas *reservation stations* quanto o necessário para que as taxas de aceleração não fossem afetadas pela falta de recursos. Assim, para cada programa da bateria de testes, foram realizadas diversas simulações, cada uma delas com um número maior de *reservation stations*, até que as taxas de aceleração ficassem estabilizadas.

Graças ao mecanismo de predição de desvios, o despacho jamais é interrompido. Diante da impossibilidade de executar mais instruções do que as que são despachadas (devido as dependências verdadeiras de dados e aos tempos de latência do despacho e da execução das instruções) é de se esperar que para janelas com maior capacidade de armazenamento (4, 8 e 16 instruções), o número de *reservation stations* ocupadas simultaneamente aumente com o tempo de execução.

Verificamos que os programas *BCDBin*, *Branch* e *Quick-Sort* precisaram ocupar simultaneamente pouco mais de 2.900, 5.000 e 1.000 das *reservation stations* associadas às unidades *Core*, respectivamente, para que suas taxas de aceleração não fossem modificadas. A necessidade de um grande número de *reservation stations* se deve ao elevado potencial de paralelismo entre as instruções deste programa. Já para o *Hu-Tucker* e o *Livermore*, embora o mecanismo de predição de desvios provocasse a ocupação simultânea de quantidades ainda maiores de *reservation stations* (se fossem providas tantas), o baixo nível de paralelismo entre suas instruções torna desnecessário empregar mais do que 64 *reservation stations* (não realizamos simulações com menos do que 64 *reservation stations*).

Um modelo de máquina com 1.000 (ou mais) *reservation stations* e que empregue um mecanismo de predição de desvios como o aqui descrito, certamente não pode ser considerado realista (com a tecnologia atualmente disponível). Contudo, esse modelo é útil na avaliação do impacto dos desvios condicionais no desempenho da máquina. As taxas de aceleração e demais medidas apresentadas nesta seção, correspondem aos limites superiores teóricos do desempenho que pode ser obtido por uma máquina que utilize um mecanismo de predição de desvios realista. Além disso, os resultados obtidos não só demonstram a importância desse mecanismo nos processadores Super Escalares como estimulam as pesquisas nesta área.

Capítulo 6

Conclusões

A detecção e exploração do paralelismo de baixo nível, foram os temas abordados ao longo dessa tese. Empregando um modelo de máquina Super Escalar —derivado do processador i860 da Intel— avaliamos o efeito de importantes detalhes arquiteturais no desempenho de algoritmos de escalonamento dinâmico de instruções.

Após termos comprovado que o algoritmo de despacho associativo (i.e., o algoritmo de Tomasulo) é mais eficiente do que os outros algoritmos, decidimos adotá-lo no nosso modelo de máquina Super Escalar.

Com o objetivo de compatibilizar esse método de escalonamento dinâmico com as características presentes nos processadores Super Escalares da atualidade, modificamos o algoritmo de Tomasulo de modo que ele levasse em consideração o despacho em paralelo de múltiplas instruções. Mecanismos responsáveis pela remoção de ambigüidades no endereçamento de palavras da memória e pela predição de desvios foram incorporados em algumas configurações do nosso modelo de arquitetura Super Escalar. Independentemente da configuração utilizada pelos nossos experimentos, a atuação do algoritmo de Tomasulo foi ampliada, tornando-se responsável pelo controle de todos os tipos de unidades funcionais.

Utilizando os resultados produzidos pela simulação de uma bateria de programas de teste, importantes parâmetros relacionados com o desempenho de uma arquitetura Super Escalar foram identificados e apresentados ao longo dessa tese.

Dentre os parâmetros arquiteturais que influenciam a efetividade de um processador Super Escalar, investigamos o impacto causado: pelo total e mistura das unidades funcionais que constituem a máquina; pelo número de instruções despachadas simultaneamente; pela quantidade de barramentos de dados (*CDBs*) etc. Variando esses parâmetros, verificamos que as taxas de aceleração correspondentes ficaram compreendidas entre 2,29 e 2,85 na maioria dos programas de teste. A taxa de aceleração varia conforme o componente da bateria de teste, e taxas de até 6,14 foram atingidas por um dos componentes.

Constatamos que é mais vantajoso dimensionar a janela de instruções de um processador Super Escalar, de modo que ela seja capaz de armazenar 2 ou 4 ítens. Janelas com maior capacidade de armazenamento resultam em ganhos insignificantes na taxa de aceleração, e por essa razão torna-se desvantajoso investir nesse parâmetro.

Sob nosso ponto de vista, a determinação da capacidade ideal da janela deve levar em consideração, também, o nível de ocupação dos recursos do *hardware* subjacente. Conforme apontado pelas medidas de desempenho apresentadas nesse trabalho, a simples elevação na taxa de aceleração deve ser melhor investigada. Usualmente, essa melhoria no desempenho de uma configuração pode resultar do grande volume de recursos que foi incorporado no processador, sem que tenhamos com isso, uma redução no nível de ociosidade dos componentes da máquina. Por exemplo, se examinarmos a porcentagem de utilização das unidades funcionais de uma configuração que apresentou um elevado grau de eficiência, podemos verificar que a taxa de ociosidade, em alguns casos, aumentou.

Tendo em vista o importante papel desempenhado pelos *CDBs*, realizamos diversos experimentos para investigar o efeito desse componente no nosso modelo de máquina. Ao permitir o início antecipado (fora de ordem) de instruções que apresentam dependências falsas, esses barramentos viabilizam um significativo incremento no número de instruções sendo executadas em paralelo. Uma outra vantagem apresentada pelos *CDBs* refere-se a redução no número de acessos aos bancos de registradores: através de uma única transferência, os valores produzidos pelas operações são propagados (via barramento) para todos dispositivos funcionais

e bancos de registradores que estiverem aguardando por esses resultados.

Durante a avaliação da influência dos *CDBs*, restringimos nossos estudos ao impacto provocado pelo emprego de múltiplos barramentos. Ao constatar que a taxa de utilização dos *CDBs* é muito baixa, (e.g., nas configurações cujas janelas armazenam quatro instruções, os *CDBs* ficaram ociosos durante pelo menos 59,72% do tempo de execução) e que o incremento nas taxas de aceleração (quando usamos múltiplos *CDBs*) é modesto, concluímos que é desnecessário utilizar mais do que um barramento: o custo de implementação de múltiplos *CDBs* é muito elevado quando comparado com o nível de utilização desses recursos e com a aceleração resultante.

Já no caso das *reservation stations*, observamos que a utilização desse tipo de componente depende da capacidade da janela de instruções. Nos processadores cujas janelas armazenam uma única instrução, a utilização de *reservation stations* não é vantajosa. Nas configurações com essa janela, as taxas de ocupação indicaram que para cada tipo de dispositivo funcional, precisamos somente de uma *reservation station* e de uma unidade funcional. Por esse motivo, as instruções poderiam ser despachadas diretamente para as unidades funcionais.

Nas configurações cujas janelas armazenam mais do que uma instrução, a presença de *reservation stations* é bastante vantajosa. No caso das *reservation stations* associadas à Unidade de Acesso à Memória, essa presença é fundamental: tendo em vista que existe somente uma unidade desse tipo (em todas as configurações do nosso modelo de máquina), então o mecanismo de despacho seria freqüentemente interrompido se uma única *reservation station* fosse empregada.

Já para as unidades do tipo *Core*, as diferenças entre as taxas de ocupação de unidades funcionais e *reservation stations* ($T_{rs} - T_{uf}$) e entre o total de dispositivos destes tipos utilizados ($RS_{core} - UF_{core}$) não são tão significativas. Contudo, a exemplo da unidade de Acesso à Memória, espera-se que para configurações mais realistas (i.e., com um número menor de unidades *Core*) tais diferenças aumentem, destacando mais uma vez a importância das *reservation stations*.

Conforme discutido no Capítulo 5, instruções de desvio condicional

provocam interrupções no mecanismo de despacho. Nossos experimentos revelaram que devido à esse tipo de instrução, o processo de despacho pode permanecer interrompido durante até 89,64% do tempo de execução. Ao longo desse intervalo de tempo, as unidades funcionais permanecerão ociosas, aguardando pela chegada de novas instruções, resultando numa queda no desempenho.

Com o objetivo de avaliar, mais precisamente, o custo imposto pelas instruções de desvio condicional, um algoritmo de predição de desvios foi introduzido no nosso modelo de arquitetura Super Escalar. Esse algoritmo é otimista, e pode ser visto como um oráculo oniciente: ele sempre sabe se a transferência de controle irá ocorrer ou não.

Por esse motivo, o mecanismo de despacho não precisa ser interrompido, e ficar aguardando pelo término da avaliação da condição (e opcionalmente pelo término da avaliação do endereço efetivo da instrução alvejada pelo desvio). Limitada somente pelas dependências verdadeiras de dados, as taxas de aceleração apresentadas por essa configuração atingiram até a 14,66 (em termos da máquina de referência).

Comparando o tempo de processamento dos modelos com e sem algoritmo de predição de desvios, constatamos crescimento nas taxas de aceleração variando de 66% a 500%. Uma vez que existe (no modelo com predição de desvios) um grande número de instruções prontas para serem executadas a cada ciclo, tornou-se mais vantajoso empregar janelas com 4 e 8 instruções. Além disso, verificamos que as unidades funcionais apresentaram uma melhor utilização. Por exemplo, as unidades *Core* das configurações cujas janelas contêm 4 instruções, permaneceram ociosas durante 20,18 % do tempo de execução (no máximo). Esse valor é 7,68 pontos percentuais menor do que a taxa de ociosidade mínima de configurações sem algoritmo de predição de desvios.

Como conseqüência da utilização do algoritmo de predição de desvios, verificamos que ocorreu um acentuado crescimento na demanda por recursos, em especial pelas *reservation stations*. Em alguns casos, fomos forçados a utilizar mais do que 5.000 *reservation stations* para que não houvesse redução, ainda que

insignificante, nas taxas de aceleração.

Os modelos de máquinas Super Escalares empregados não podem ser considerados realistas. Certas limitações impostas pelo estado atual da tecnologia (e.g., conflitos no acesso aos bancos de registradores, emprego de grande quantidade de recursos) foram desconsiderados. Além disso, mecanismos de predição de desvios que sejam sempre capazes de avaliar corretamente com antecedência o destino de um dessvio condicional, não existem. Portanto, as taxas de aceleração e demais medidas obtidas correspondem aos limites superiores teóricos do desempenho que pode ser obtido por modelos de máquinas Super Escalares semelhantes, porém com uma configuração mais próxima do que é factível.

Para dar prosseguimento ao trabalho aqui apresentado, pretendemos extrair de um modelo realista, medidas semelhantes. Este modelo deverá contar com poucos recursos (*reservation stations*, unidades funcionais e *CDBs*). A variação no número de recursos empregados permitirá que seja avaliada a importância deles sobre as taxas de acelaração e de ocupação. Comparando os dados apresentados nesta tese com os que serão obtidos a partir destes novos experimentos, será possível identificar "gargalos" que limitem o desempenho da máquina. As pesquisas subseqüentes deverão buscar formas de eliminar tais "gargalos" e assim, aproximar o desempenho dos limites superiores teóricos.

Referências Bibliográficas

[ACOS86] R. D. Acosta, J. Kjelstrup e H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," IEEE Transactions on Computers, Vol. C-35, No. 9, September 1986, pp. 815-828.

[AUHT85] A. K. Uht, "Hardware Extraction of Low Level Concurrency from Sequential Instruction Streams," Ph.D. Thesis, Carnegie-Mellon University, December 1985.

[BAR92a] F. M. B. Barbosa e E. S. T. Fernandes, "Dispatching Simultaneous Instructions," Microprocessing and Microprograming, The Euromicro Journal, North Holland (Editor), Vol. 34, No. 1-5, February 1992, pp. 227-230.

[BAR92b] F. M. B. Barbosa e E. S. T. Fernandes, "Associative Dispatch of Multiple Instructions: A Study of its Effectiveness," Preliminar Version, Technical Report, Programa de Sistemas e Computação, ES-263/92, COPPE/UFRJ, June 1992, 18 pages.

[BOLA67] L. J. Boland, G. D. Granito, A. U. Marcotte, B. U. Messina, J. W. Smith, "The IBM System/360 Model 91: Storage System," IBM Journal, No. 11, 1967, pp. 54-68.

[BUTL91] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales e M. Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991, pp. 276-286.

[CHAN91] P. P. Chang, W. Y. Chen, S. A. Mahlke e W. W. Hwu, "Comparing Static and Dynamic Code Scheduling for Multiple Instruction Issue Pro-

cessors," Proceedings of the 24th Annual International Symposium on Microarchitecture, MICRO-24, November 1991, pp. 25-33.

[CONT65] S. D. Conte, "Elementary Numerical Analysis," McGraw-Hill Book Co., New York, NY, USA, 1965.

[DAND67] D. W. Anderson, F. J. Sparacio e R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," IBM Journal, No. 11, 1967, pp. 8-24.

[DIEF92] K. Diefendorff e M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," IEEE Micro, April 1992, pp. 40-63.

[DITZ87] D. R. Ditzel e H. R. Mclellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proceedings of the 14th Annual International Symposium on Computer Architecture, 1987, pp. 2-9.

[DWYE87] H Dwyer e H. C. Torng, "A Fast Instruction Dispatch Unit for Multiple and Out-of-Sequence Issuances," Technical Report EE-CEG-87-15, School of Electrical Engineering, Cornell University, Ithaca, NY, USA, November 1987, 23 pages.

[FER92a] E. S. T. Fernandes e F. M. B. Barbosa, "Effects of Building Blocks on the Performance of Super-Scalar Architectures," Preliminar Version, Technical Report, Programa de Sistemas e Computação, ES-261/92, COPPE/UFRJ, January 1992, 18 pages.

[FER92b] E. S. T. Fernandes e F. M. B. Barbosa, "Effects of Building Blocks on the Performance of Super-Scalar Architectures," Proceedings of the 19th Annual International Symposium on Computer Architecture, ACM-SIGARCH and IEEE Computer Society, Australia, May 19-21, 1992, pp. 36-45.

[FISH84] J. A. Fisher, "VLIW Machine: A Multiprocessor for Compiling Scientific Code," IEEE Computer, July 1984, pp. 45-53.

[FLYN67] M. J. Flynn, "The IBM System/360 Model 91: Some Remarks on System Development," IBM Journal, No. 11, 1967, pp. 2-7.

[FORS67] G. Forsythe e C. B. Moler, "Computer Solution of Linear Algebraic Systems," Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[GROH90] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," IBM J. Res. Develop., Vol. 34, No. 1, January 1990, pp. 37-58.

[HINT89] G. Hinton, "80960 — Next Generation," Proceedings of the 34th COMPCON, IEEE, San Francisco, CA, USA, March 1989, pp. 13–17.

[INTE89] Intel, "i860 64-Bit Microprocessor Programmer's Reference Manual," Intel, 1989.

[JLEE84] J. K. F. Lee e A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, January 1984, pp. 6-21.

[KNUT73] D. Knuth, "Art of Computer Programming," Addison-Wesley Publishing Co., USA, 1973.

[KELL75] R. M. Keller, "Look-Ahead Processors," Computing Surveys, Vol. 7, No. 4, December 1975, pp. 177-195.

[KRIC91] R. F. Krick e A. Dollas, "The Evolution of Instruction Sequencing," IEEE Computer, April 1991, pp. 5-15.

[McGE90] S. McGeady, "Inside Intel's i960CA Superscalar Processor," Microprocessors and Microsystems, Vol. 14, No. 6, July/August 1990, pp. 385-396.

[MCMA83] F. H. McMahon, "Fortran Kernels: MFLOPS," Lawrence Livermore National Laboratory, 1983.

[PATT85] Y. N. Patt, H. Hwu e M. C. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," Proceedings of the 18th International Microprogramming Workshop, December 1985, pp. 103-108.

[PLES88] A. R. Pleszkun e G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors," Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp. 37-44.

[RAMA77] C. V. Ramamoorthy e H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No. 1, March 1977, pp. 61-102.

[SAND67] S. F. Anderson, J. G. Earle, R. E. Goldschmidt e D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," IBM Journal, No. 11, 1967, pp. 34-53.

[SMIT82] J. Smith, "Decoupled Access / Execute Computer Architectures," Proceedings of the 9th International Symposium on Computer Architecture, April 1982, pp. 113-119.

[THOR64] J. E. Thornton, "Parallel Operation in the Control Data 6600," AFIPS Proceedings FJCC, pt 2, Vol. 26, 1964, pp. 33-40.

[TJAD70] G. S. Tjaden e M. Flynn, "Detection and Parallel Execution of Independent Instructions," IEEE Transactions on Computers, Vol. C-19, No. 10, October 1970, pp. 889-895.

[TJAD73] G. S. Tjaden e M. Flynn, "Representation of Concurrency with Ordering Matrices," IEEE Transactions on Computers, Vol. C-22, No. 8, August 1973, pp. 752-761.

[TOMA67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal, No. 11, 1967, pp. 25-33.

[WARR90] H. S. Warren Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," IBM J. Res. Develop., Vol. 34, No. 1, January 1990, pp. 85-92.

[WEIS84] S. Weis e J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," Proceedings of the 11th Annual International Symposium on Computer Architecture, June 1984, pp. 110-118.

[YOHE72] J. M. Yohe, "Hu-Tucker Minimum Redundancy Alphabetic Coding Method [Z]," CACM, Vol. 15, No. 5, May 1972, pp. 360-362.